Comprando esta edição tenha acesso na WEB a:
+210 vídeo-aulas e +140 artigos

www.clubedelphi.net **Feita para desenvolvedores**

# ClubeDelphi

Ano 6 · Edição 78 · R$8,90

DevMedia GROUP

ISSN 1517990-7

**ASP.NET**
Editando, incluindo e excluindo registros no DataGrid

**Conheça em detalhes as principais classes da VCL do Delphi**

**Mão na massa**
Criando formulários dinamicamente apartir de registros de um BD

**Testes**
Desenvolvimento Ágil e Seguro com Test-Driven Development (TDD)

# FireBird/ Interbase

## Crie um controle de estoque com SPs e triggers

## Especial

**Ferramentas de análise para IB/FB**
faça um ckeck-up no seu banco de dados, encontre erros, otimize instruções SQL, gere planos de execução e aumente a performance da sua aplicação Delphi

**Mini Curso Delphi e UML**
Construa um sistema usando

**Coluna: Ask the Expert!**
Como capturar o IP externo dinamicamente,

# Quantas cópias de seu software existem no mercado?

## Esta é a chave mais segura do mundo contra pirataria de software.

## Comprove você mesmo!

SafeNet®
*The Foundation of Information Security*

Alameda Tocantins, 280 - Barueri-SP

Tel: +55 11 4208-7700

http://br.safenet-inc.com - Contato: infobrasil@safenet-inc.com

NÃO PERCA

**PORTAL DO ASSINANTE**

A ClubeDelphi tem uma novidade para você que comprou este exemplar na banca de jornal: você pode acessar GRATUITAMENTE, o Portal do Assinante ClubeDelphi!

**Confira o que você encontra no Portal do Assinante:**

- Mais de 180 Vídeo Aulas!
- Diversos Mini-Cursos Gratuitos!
- 1 Livro Eletrônico sobre ADO.NET e BDP!
- Mais de 130 Artigos Exclusivos!

Para Utilizar o Portal do Assinante, acesse www.devmedia.com.br/clubedelphi/portal.asp e utilize as informações abaixo:

**Login:** MVD.CD

**Senha: A78LY**

O acesso é válido por 30 dias a partida da data de lançamento da revista. Todos os meses a ClubeDelphi lhe dará uma senha válida para acessar o portal. Comprando a revista regularmente em bancas, você terá acesso ininterrupto a ele!

**Corpo Editorial**

**Diretor Editorial e Administrativo**
Gladstone Matos
gladstone@neoficio.com.br

**Designer**
Vinicius O. Andrade
viniciusoandrade@gmail.com

**Ilustrador**
Bruno Liserra
liserra3@hotmail.com

**Capa**
Antonio Xavier
jaxs_design@yahoo.com.br

**Articulistas desta edição**
Rodrigo Sendin, Michael Benford, Paulo Roberto Quicoli, Everson Borges Volaco, Jorge Luis Bublitz, Danilo Gazoli Resende e Gustavo Chaurais

**Editor Geral**
Guinther Pauli
guinther@devmedia.com.br

**Editor Técnico**
Luciano Pimenta
lucianopimenta@clubedelphi.net

**Revisão**
Fabio Correa
fabiocorrea@devmedia.com.br

**Distribuição**
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

**Publicidade**
Para informações sobre veiculação de anúncio na revista ou no site entre em contato com:
Kaline Dolabella
publicidade@devmedia.com.br

**Atendimento ao Leitor**

**A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:**

Thiago Andrade – Atendimento ao Leitor
www.devmedia.com.br/central/default.asp
(21) 2220-5375

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 2220-5375

**Parcerias**

**Para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:**
Jeff Wendell
jeff@devmedia.com.br

ASSESPRO PARANÁ

**Fale com o Editor!**

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão! Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

Luciano Pimenta - Editor do Site
lucianopimenta@clubedelphi.net

**ClubeDelphi**

**Ano 6 - 78ª Edição - 2006 - ISSN 1517990-7** **Impresso no Brasil**

# Editorial

Nesta edição você aprenderá importantes técnicas para otimizar sua aplicação de banco de dados. No artigo do Jorge, aprenda a implementar funcionalidades, regras de negócio e cálculos no próprio servidor de banco de dados IB/FB. Com isso, temos uma aplicação mais flexível, com menor impacto na mudança de uma funcionalidade, que faz melhor uso do hardware servidor e do tráfego de dados na rede. Por exemplo, uma regra armazenada no servidor pode ser alterada sem a necessidade de recompilação do sistema e consequentemente deploy para todas as estações clientes. Além disso, o SGBD é capaz de otimizar a execução de Stored Procedures, através da análise de planos de execução e pré-compilação das instruções. Não estamos falando só de aumento de performance, mas também da segurança da solução como um todo.

Ainda sobre otimizações, o Everson apresenta duas importantes ferramentas para análise de bancos IB/FB, que permitem vasculhar toda a estrutura do BD procurando por inconsistências, erros, possíveis otimizações a serem feitas em índices, stored procedures, triggers, além de permitirem a análise de planos de execução, monitoramento da execução de instruções SQL, geração de estatísticas e muito mais.

Ainda nesta edição, muito sobre ASP.NET, ADO.NET, UML, VCL, geração automática de formulários usando técnicas avançadas de RTTI, desenvolvimento ágil e seguro com Test-Driven Development.

Boa leitura e um grande abraço!

**Guinther Pauli –** *guinther@devmedia.com.br*
**MCP, MCAD, MCSD.NET, Delphi Certified**

## ÍNDICE

**04 -** Editando, Incluindo e Excluindo registros no DataGrid
*Rodrigo Sendin*

**12 -** Mini-Curso de Delphi e UML – Parte III
*Paulo Roberto Quicoli*

**20 -** VCL Revelada
*Michael Benford*

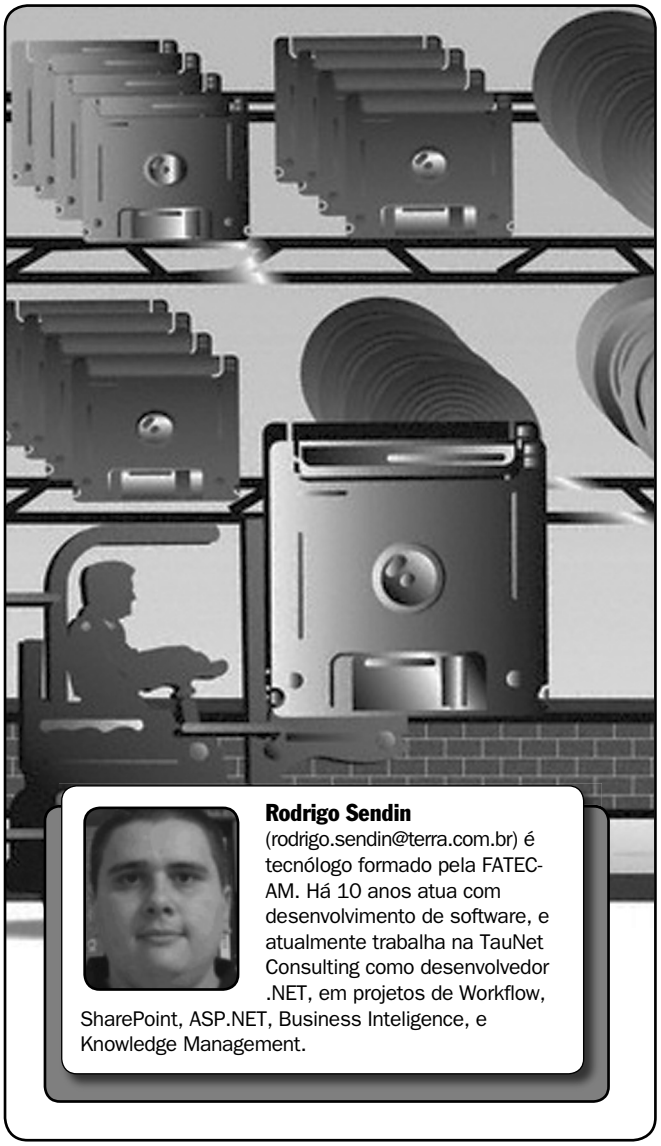
**26 -** Análise, reparação e otimização de bancos IB/FB
*Everson Borges Volaco*

**32 -** Criando formulários dinamicamente a partir de registros em um BD
*Danilo Gazoli Resende*

**38 -** Desenvolvimento Ágil e Seguro com Test-Driven Development
*Gustavo Chaurais*

**46 -** Uma abordagem prática de Stored Procedures e Triggers
*Jorge Luis Bublitz*

# Editando, Incluindo e Excluindo registros no DataGrid



**Rodrigo Sendin**
(rodrigo.sendin@terra.com.br) é tecnólogo formado pela FATEC-AM. Há 10 anos atua com desenvolvimento de software, e atualmente trabalha na TauNet Consulting como desenvolvedor .NET, em projetos de Workflow, SharePoint, ASP.NET, Business Inteligence, e Knowledge Management.

Na edição 73 da ClubeDelphi, vimos como trabalhar com o *DataGrid* no ASP.NET. Pudemos conferir as principais funcionalidades desse controle, e como é simples expor dados em grid com o ASP.NET. Neste artigo veremos algumas *features* do *DataGrid* que permitirão adicionar funcionalidades de edição, exclusão e inclusão de registros.

## Assistende de configuração do FbDataAdapter

Para os exemplos deste artigo crie um projeto ASP.NET no Delphi 8, 2005 ou 2006, e na página *WebForm1.aspx* inclua um *DataGrid*. Se vocês lembram do último artigo, preenchemos o *DataGrid* com os dados da tabela *Products* do banco *Northwind* do SQL Server.

Fizemos isso via código, criando os objetos necessários do ADO.NET para buscar os dados na tabela e preencher o grid. Neste exemplo vamos fazer diferente, vamos utilizar o provider ADO.NET do próprio banco de dados Firebird (que é mais otimizado para acesso ao FB), juntamente com alguns assistentes que ajudarão a preencher o *DataGrid* com os dados da tabela *Customer*, e veremos como é simples e nos poupa um bocado de código. Adicione um *FbDataAdapter* (**Figura 1**) na página.

### CLUBEDELPHI PLUS!

*Acesse agora o mesmo o portal do assinante ClubeDelphi (www.clubedelphi.net/portal) e assista a uma vídeo-aula de Luciano Pimenta que mostra como atualizar dados de um DataGrid usando FbCommands.*

*http://www.devmedia.com.br/articles/viewcomp.asp?comp=2050*

### Instalando o Firebird .NET Provider

Baixe o arquivo do endereço *www.firebirdsql.org/index.php?op=files&id=netprovider* (para a versão 1.1 do .NET). Após a instalação, para adicionar os componentes no Delphi, acesse o menu *Component>Installed Components*, clique no botão *Select an Assembly* e escolha o arquivo *FirebirdSql.Data.Firebird.dll* no diretório *C:\Arquivos de programas\FirebirdNETProvider[versão]*.

Veja que o assistente de configuração do *FbDataAdapter* foi iniciado (caso isso não ocorra, clique com o botão direito do mouse no componente e escolha a opção *Configure Data Adapter*). Essa é apenas uma tela de apresentação, clique em *Next*. Na tela a seguir precisamos criar uma nova conexão com o banco, ou se você tiver um *FbConnection* na página, basta indicá-lo.

Para criar uma conexão com o *Employee.fdb* do Firebird, clique em *New Connection* e informe o nome do *Servidor*, local do arquivo FDB, usuário e senha se necessário (**Figura 2**). Em seguida clique em *Accept* e novamente em *Next*.

**Figura 1.** Provider do Firebird instalado na Tool Palette

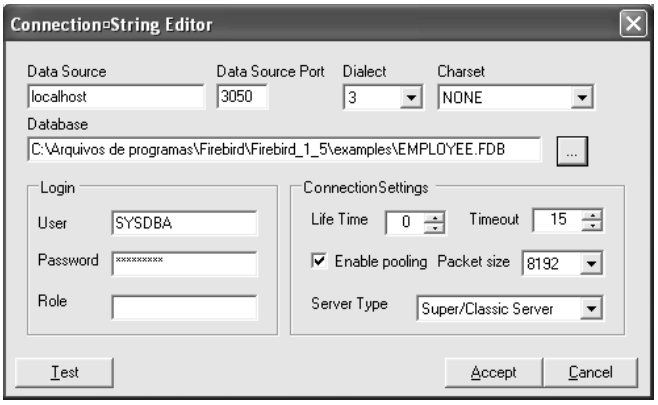



**Figura 2.** Criando uma conexão com o banco Employee.fdb do Firebird

Estamos agora na terceira tela do assistente, onde devemos definir como os comandos SQL serão armazenados. Podemos usar os comandos SQL (*Select, Insert, Update, Delete*) ou podemos utilizar *Stored Procedures* existentes (**Figura 3**).

Para este exemplo vamos simplesmente criar os comandos, mantendo a primeira opção selecionada e clicando em *Next*. Na próxima etapa do assistente precisamos definir o comando SELECT que utilizaremos para recuperar os dados necessários e preencher o *DataGrid*. Vamos selecionar todos os campos da tabela *Customer*, para isso basta inserir o comando "SELECT * FROM Customer" no espaço reservado para o mesmo (**Figura 4**).

Também podemos utilizar a opção *Generate Insert, Update and Delete statements*, para indicar se o assistente irá ou não criar os comandos *Insert*, *Update* e *Delete*. Marque a opção e clique em *Next*. Observe que essa é a última tela do assistente e mostra que, neste exemplo, os comandos SQL foram criados com sucesso, bastando agora clicar em *Finish*.

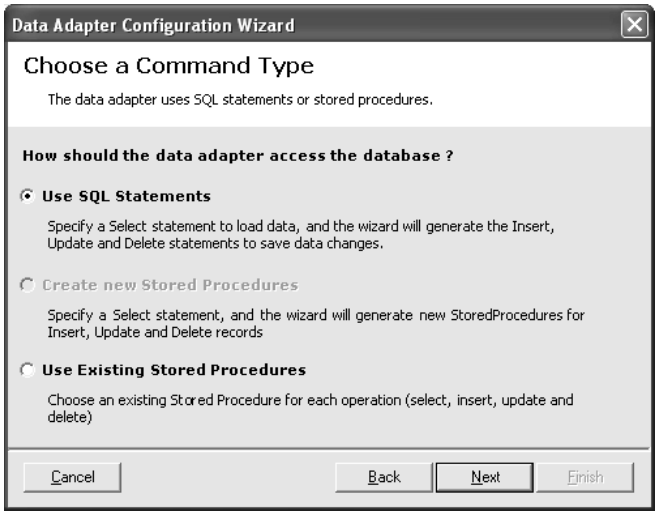



**Figura 3.** Definindo como os comandos SQL serão gerados

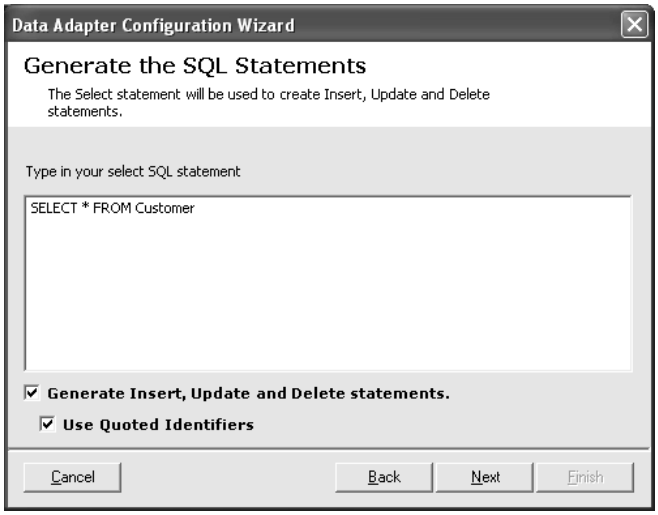



**Figura 4.** Definindo comando Select para retornar os registros da tabela Customer

## Gerando o DataSet

Veja que após finalizar o assistente, dois novos objetos foram criados na parte inferior da página. O primeiro é o *FbDataAdapter* que foi chamado de *FbDataAdapter1*, e o segundo é o *FbConnection1*, que é uma instância da classe *FbConnection*, responsável por realizar a conexão com o banco.

Como vimos no último artigo sobre o *DataGrid*, para preencher o controle com dados de uma tabela, precisamos de um *DataSet* (também podemos usar um *FbCommand* chamando o método *ExecuteReader*). Para criar o *DataSet*, clique com o botão direito sobre o *FbDataAdapter1* e escolha a opção *Generate DataSet*.

Podemos utilizar um *DataSet* já existente ou criar um novo. Escolha a opção *New* e digite "dsCustomer". Observe também que o *FbDataAdapter* mostra todas as tabelas, para que você escolha a que terá os dados retornados. Para adicionar o comando SQL, presente no *Select* do *FbDataAdapter1*, marque a opção *Use DataAdapter Select command* (**Figura 5**). Clique em OK e veja que um *DataSet* foi adicionado na parte inferior na página.

Basta agora definir o *DataSource* do *DataGrid*, que deverá apontar para o *DataSet* que criamos. Clique com o botão direito sobre o *DataGrid* e escolha a opção *Property Builder*. No campo *DataSource* escolha o *dsCustomer*, em *DataMember* escolha *Table* e em *Data key field* selecione CUST_NO (**Figura 6**).

Ao clicar no botão OK você verá que o *DataGrid* já mostrará os campos da tabela. Aproveite e defina um layout para o *DataGrid* através da opção *Auto Format* que temos ao clicar com o botão direito sobre o *DataGrid* (neste exemplo foi utilizado o formato *Professional 1*).

Para finalizar essa primeira etapa, vamos incluir um código no *Page_Load*, onde estamos simplesmente preenchendo o *DataSet* através do método *Fill* do *FbDataAdapter*. Em seguida chamamos o método *DataBind* do *DataGrid* para expor os dados. Por último armazenamos nosso *DataSet* em uma variável de sessão.

Observe que estamos executando essas ações apenas quando a página não estiver sofrendo um *PostBack*, senão estamos recuperando o *DataSet* da variável de sessão, assim selecionando os dados no banco apenas uma vez. Veja o código na **Listagem 1**.

**Listagem 1. Código para preencher DataSet e expor dados no DataGrid**

```
var
  aPK: array[0..1] of DataColumn;
begin
  if not IsPostBack then
  begin
    FbDataAdapter1.Fill(dsCustomer);
    aPK[0] :=
      dsCustomer.Tables['Table'].Columns['CUST_NO'];
    dsCustomer.Tables['Table'].PrimaryKey := aPK;
    DataGrid1.DataBind;
    Session['Customer'] := dsCustomer;
  end
  else
    dsCustomer:= Session['Customer'] as DataSet;
```

Criamos uma variável do tipo *DataColumn* para especificar qual a chave primária da tabela. Isso será útil na atualização do registro, que veremos adiante. Salve, compile, execute e confira o resultado (**Figura 7**).

A partir daqui você pode implementar as funcionalidades que vimos no artigo de *DataGrid* da edição 73, dentre elas podemos citar: formatações das *BoundColumns*, paginação, ordenação, colunas template etc.

## Editando registros no DataGrid

Vamos criar a funcionalidade de edição de registros no *DataGrid*. Clique com o botão direito sobre o *DataGrid*, escolha a opção *Property Builder* e vá na aba *Columns*. Temos uma área chamada *Available Columns*, onde temos vários tipos possíveis de colunas. Vamos expandir a opção *Button Column*.

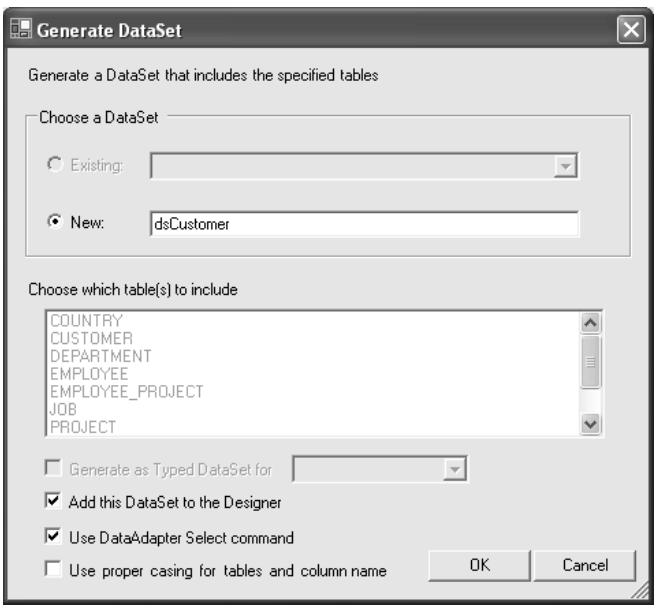



**Figura 5.** Criando um novo DataSet através do FbDataAdapter

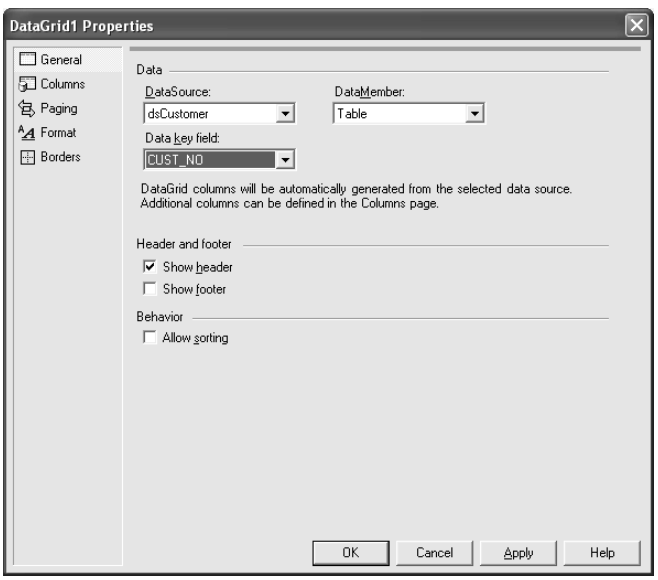



**Figura 6.** Definindo o DataSource do DataGrid

Escolha o segundo item, chamado *Edit, Update, Cancel* e clique no botão para adicionar a coluna, representado por um sinal de maior ( > ). A nova coluna será incluída, e como podemos ver, podemos editar algumas propriedades dessa coluna, como por exemplo, os textos de cada um dos botões.

A única propriedade modificada no exemplo foi o *Button Type*, que foi definido como *Push Button* (**Figura 8**). Clique em OK e veja que a nova coluna foi adicionada ao *DataGrid*.

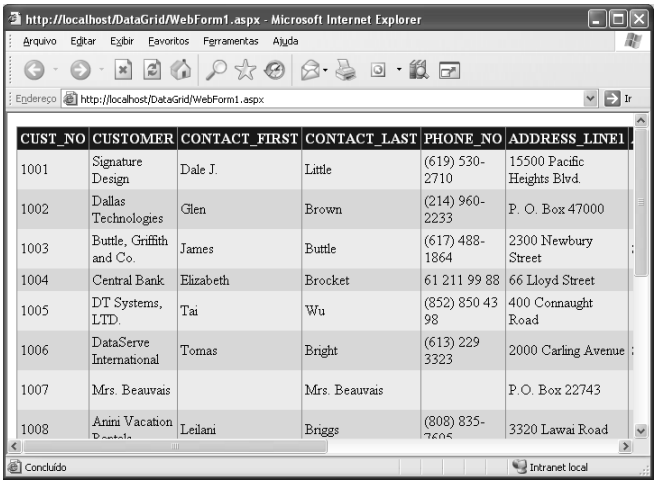
Salve, compile e execute o seu projeto. Você verá que o botão aparece, mas ao clicarmos nele nada acontece. Precisamos codificar algumas ações para habilitar a funcionalidade de edição dos registros. Volte ao design da página, clique no *DataGrid* e na janela de propriedades modifique para a aba de eventos.

Veja que temos uma categoria chamada *Action* que contém todos os eventos que disparam uma ação no grid. Vamos dar um duplo clique no evento *EditCommand* e adicionar o seguinte código:

```
DataGrid1.EditItemIndex := e.Item.ItemIndex;
DataGrid1.DataBind();
```

Veja que na primeira linha estamos alterando a propriedade *EditItemIndex* do *DataGrid* com o índice do item que foi selecionado para edição. Ao atualizar essa propriedade, a linha será modificada para que possamos alterar os dados do registro. Veja também que estamos novamente chamando o método *DataBind* para que os dados sejam atualizados no *DataGrid,* inclusive na linha que estiver sendo editada.

Apenas isso não basta para que as alterações sejam feitas no registro. Para isso precisamos codificar o evento *UpdateCommand* do *DataGrid*. Volte ao designer da página, inclua um *Label* abaixo do *DataGrid* e apague o conteúdo da propriedade *Text*. Em seguida clique no *DataGrid*, dê um duplo clique sobre o evento *UpdateCommand* e adicione o código da **Listagem 2**.




| CUST_NO | CUSTOMER | CONTACT_FIRST | CONTACT_LAST | PHONE_NO | ADDRESS_LINE1 |
|---|---|---|---|---|---|
| 1001 | Signature Design | Dale J. | Little | (619) 530-2710 | 15500 Pacific Heights Blvd. |
| 1002 | Dallas Technologies | Glen | Brown | (214) 960-2233 | P. O. Box 47000 |
| 1003 | Buttle, Griffith and Co. | James | Buttle | (617) 488-1864 | 2300 Newbury Street |
| 1004 | Central Bank | Elizabeth | Brocket | 61 211 99 88 | 66 Lloyd Street |
| 1005 | DT Systems, LTD. | Tai | Wu | (852) 850 43 98 | 400 Connaught Road |
| 1006 | DataServe International | Tomas | Bright | (613) 229 3323 | 2000 Carling Avenue |
| 1007 | Mrs. Beauvais | | Mrs. Beauvais | | P.O. Box 22743 |
| 1008 | Anini Vacation | Leilani | Briggs | (808) 835- | 3320 Lawai Road |

**Figura 7.** DataGrid preenchido com os registros da tabela Customer

**Listagem 2. Gravando alterações no DataSet**

```
var
  aRow: DataRow;
begin
  { Procura no DataSet o registro para edição.
   Utilizamos a propriedade DataKeys do DataGrid,
   para capturar a chave primária, especificada na
   propriedade DataKeyField }
  aRow := dsCustomer.Tables[0].Rows.Find(
    DataGrid1.DataKeys[e.Item.ItemIndex]);
  try
    { Alterando os campos da tabela,
      através dos controles de edição }
    aRow['CUSTOMER'] :=
      (e.Item.Cells[2].Controls[0] as TextBox).Text;
    aRow['CONTACT_FIRST'] :=
      (e.Item.Cells[3].Controls[0] as TextBox).Text;
    ... { Outros campos }
    { Gravando as alterações na variável de sessão }
    Session['Customer'] := dsCustomer;
    DataGrid1.EditItemIndex := -1;
    DataGrid1.DataBind;
  except
    Label1.Text := 'Erro';
  end;
end;
```

Esse é o código necessário para gravar no *DataSet* as alterações feitas no registro. Primeiro, buscamos uma referência à linha do *DataSet* que alteraremos. Veja que recuperamos os campos editados em controles *TextBox*, através da coleção *Controls* de cada uma das células do *DataGrid*.

Podemos identificar qual é a linha do *DataSet* através da propriedade *ItemIndex* do *DataGridItem*, onde poderíamos usar também o *DataSetItem*. Na seqüência estamos atualizando cada um dos atributos do registro na linha do *DataSet*.

Após a atualização, estamos gravando o *DataSet* novamente na variável de sessão e saindo do modo de edição, setando a propriedade *EditItemIndex* do *DataGrid* para -1. E novamente chamamos o método *DataBind* para que as alterações sejam apresentadas no grid. Todo esse código está inserido em uma estrutura *try/except*, assim, caso ocorra alguma exceção, veremos a mensagem no *Label* inserido na página.

E para finalizar, precisamos apenas codificar o evento do botão *Cancel*. Volte ao *design*, clique no *DataGrid*, e no evento *CancelCommand* adicione o seguinte código, que

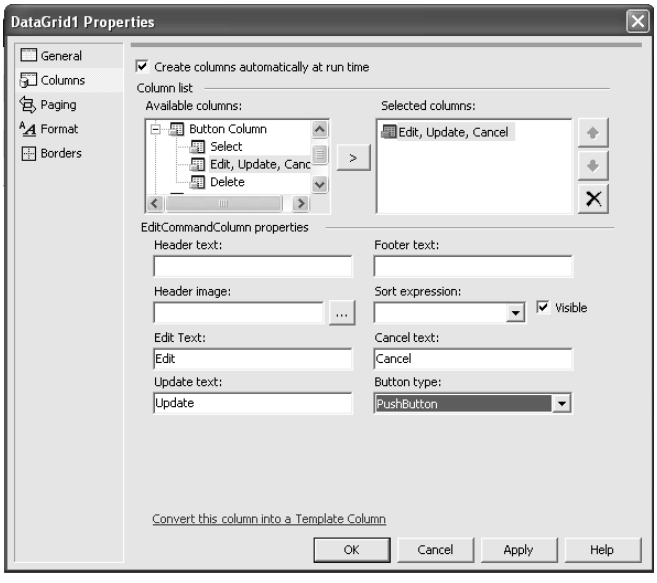



**Figura 8.** Incluindo os botões para edição dos registros

simplesmente tira o registro do modo de edição, sem realizar nenhuma atualização:

```
DataGrid1.EditItemIndex := -1;
DataGrid1.DataBind();
```

Salve, compile e execute o projeto. Clique no botão *Edit* de um registro qualquer, e veja que a linha é modificada, mostrando um *TextBox* em cada uma das colunas. Altere o nome do cliente e clique em *Update* (**Figura 9**). Veja que a alteração foi gravada no registro. Teste também o botão *Cancel* e note que ele descarta as alterações.

## Trabalhando desconectado do banco

Feche o browser e execute novamente o seu projeto. Onde estão as alterações feitas? É isso mesmo, elas foram perdidas, pois as alterações foram feitas apenas na memória, no *DataSet*, e não foram persistidas no banco.

No ADO.NET podemos trabalhar de duas maneiras: desconectados ou conectados ao banco. São dois conceitos que determinam quando persistiremos as alterações no banco. No nosso exemplo estamos trabalhando desconectados, isso quer dizer que podemos realizar inclusões/alterações/exclusões em diversos registros e só no final persistiremos essas alterações no banco, todas de uma vez.

Se trabalhássemos conectados, teríamos que persistir as alterações no momento em que elas ocorrem no registro, ou seja, neste exemplo, em vez do código inserido na **Listagem 2**, teríamos que incluir um código para disparar um comando *Update* no banco. Assim a alteração seria prontamente atualizada no BD, e não somente no *DataSet*.

Como você já deve estar imaginando, essas são duas possibilidades que atendem a necessidades distintas, portanto, quando for implementar funcionalidades de edição em *DataGrids*, avalie a realidade da aplicação e opte por trabalhar desconectado ou conectado ao banco, ponderando os prós e contras de cada uma das técnicas.

Neste exemplo trabalharemos desconectados e no final, implementaremos uma rotina que persistirá todas as alterações realizadas no *DataSet*.

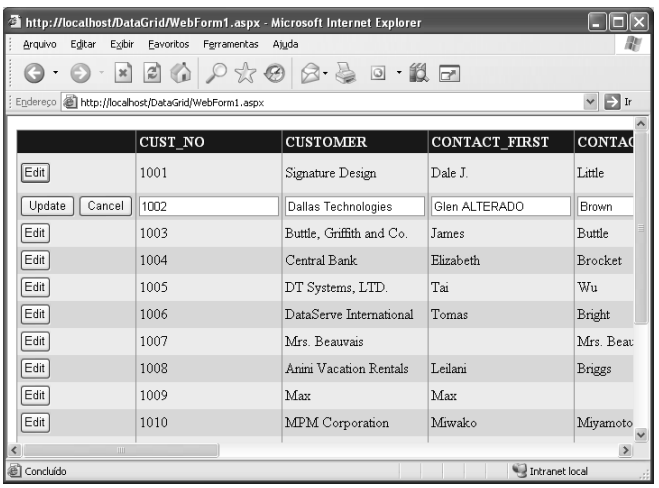



**Figura 9.** Editando registros no DataGrid

## Excluindo registros no DataGrid

Vamos implementar agora a funcionalidade de exclusão de registros. Volte ao designer da página, clique com o botão direito sobre o *DataGrid* e escolha a opção *Property Builder*. Clique na aba *Columns*, e a exemplo do que fizemos com os botões de edição, expanda a opção *Button Column*, clique em *Delete* e adicione essa nova coluna ao *DataGrid* ( > ). Modifique o *Button Type* para *Push Button* e clique em OK. Seu *DataGrid* tem agora uma coluna com o botão *Delete*. Clique no *DataGrid* e no evento *DeleteCommand* adicione o código da **Listagem 3**.

**Listagem 3. Código para exclusão dos itens no DataSet**

```
var
  aRow: DataRow;
begin
  aRow := dsCustomer.Tables[0].Rows.Find(
    DataGrid1.DataKeys[e.Item.ItemIndex]);
  if aRow <> nil then
    aRow.Delete;
  Session['Customer'] := dsCustomer;
  DataGrid1.DataBind();
end;
```

Observe que simplesmente estamos buscando o registro no *DataSet*, através de um *DataRow* e chamando o método *Delete* da respectiva linha. O método *Delete* se encarrega de excluir o registro da *DataTable*. Em seguida estamos gravando novamente o *DataSet* na variável de sessão, e atualizando o grid com o método *DataBind*.

Uma outra modificação importante que deve ser feita é no evento *UpdateCommand*. Como podemos conferir na **Listagem 2**, estamos recuperando os valores editados através da coleção *Cells*. Como incluímos a coluna com o botão *Delete* no início do *DataGrid*, os índices das células foram modificados, portanto onde estamos nos referindo a *Cells[2]*, precisamos trocar para *Cells[3]* e assim por diante.

Feita as modificações, salve, compile e execute o seu projeto. Clique no botão *Delete* e veja que o registro é excluído do *DataSet* e consequentemente do *DataGrid*. Como estamos trabalhando desconectados do banco, a exclusão foi feita apenas no *DataSet* que está na memória, e não diretamente na tabela *Customer* do banco.

Se estivéssemos trabalhando conectados ao banco, deveríamos trocar o código da **Listagem 3** para executar um comando *Delete* na tabela *Customer*. Outra observação importante é que o registro foi excluído sem nenhuma confirmação do usuário, com um pouco de *JavaScript* você pode implementar uma confirmação de exclusão do registro.

> **Dica:** Existe um controle chamado MessageBox que emite caixas de mensagens e de confirmação com JavaScript. O componente é free e assim temos muita produtividade para as aplicações ASP.NET. Para maiores informações, veja a edição 75 da ClubeDelphi e uma vídeo aula de Luciano Pimenta em www.devmedia.com.br/visualizacomponente.aspx?comp=1405&site=3.

## Incluindo registros no DataGrid

Vamos agora criar a rotina de inclusão de registros no grid. Essa é uma opção que não encontraremos na seção *Button Column* do *DataGrid*, mas é bem simples de implementar. Volte ao designer e inclua um *Button* na página, que pode ser posicionado abaixo do *DataGrid.* Esse *Button* deverá chamar-se "btnIncluir", e em *Text* digite "Incluir Cliente". Dê um duplo clique no botão e inclua o código da **Listagem 4**.

**Listagem 4. Incluindo um registro no DataSet**

```
var
  aRow: DataRow;
begin
  aRow := dsCustomer.Tables[0].NewRow();
  aRow['CUST_NO'] := &Object(1016);
  aRow['CUSTOMER'] := '';
  aRow['CONTACT_FIRST'] := '';
  ... { Outros campos }
  dsCustomer.Tables[0].Rows.Add(aRow);
  Session['Customer'] := dsCustomer;
  DataGrid1.DataBind();
  DataGrid1.EditItemIndex := DataGrid1.Items.Count -1;
  DataGrid1.DataBind();
end;
```

Veja que estamos utilizando o método *NewRow* do *DataTable*, assim podemos preencher normalmente os valores dos campos, que no caso estão sendo inclusos com valores em branco para *strings* e zero para valores numéricos. Note que no campo chave adicionei um valor arbitrário para facilitar nossos testes, mas quando for aplicar a técnica em um sistema real, lembre-se de usar *Generators* ou outra forma apropriada para obter valores para chaves-primárias.

Em seguida, chamamos o método *Add* da coleção de *Rows* do *DataTable*, passando como parâmetro a variável. Por fim, gravamos novamente o *DataSet* na variável de sessão e chamamos o método *DataBind* para que o novo registro apareça no grid.

Veja que na seqüência estamos configurando a propriedade *EditItemIndex* para que o usuário possa editar o registro que acabamos de incluir. Como o registro é incluído por último, podemos identificar o seu índice através da propriedade *Count* - 1. Novamente precisamos chamar o *DataBind* para que os dados sejam atualizados com a linha em modo de edição.

Vamos fazer um teste? Salve, compile e execute o projeto. Clique no botão *Incluir Cliente* e vá até a última linha do grid. Como mostra a **Figura 10**, veja que uma nova linha foi incluída no grid, basta o usuário preencher os campos com os valores do novo cliente e clicar em *Update* para gravar o registro no *DataSet*.

Você poderá observar que se clicar no botão *Cancel,* o registro continuará aparecendo em nosso *DataGrid*. Para fazer com que o registro seja descartado caso o usuário cancele a inclusão, seria necessário criar uma variável de controle e realizar uma verificação no evento *CancelCommand*.

## Persistindo as alterações no banco

Para finalizar, precisamos agora criar a rotina que persistirá todas as inclusões/alterações/exclusões feitas no *DataSet*. Volte ao designer da página e inclua um novo botão, que deverá chamar-se "btnGravar" e seu *Text* deve ser modificado para "Gravar Alterações". Dê um duplo clique no botão para adicionar o código que persistirá os dados no banco.

Quanto código você acha que será necessário para persistir os dados? Parece ser uma operação complexa não é? Mas em apenas uma única linha de código (mais o tratamento de execeção) resolvemos essa questão. Inclua o código da **Listagem 5** no evento gerado para o click.


Livros de Delphi

Delphi - Implementação de Algoritm
Técnicas p/ Ambientes Visuais
Visual Books
R$ 62,00

Delphi: para Todas as Versões
Ciência Moderna

R$ 29,00

Estudo Dirigido de Delphi 2005
Érica
R$ 78,50

Delphi 2005: Aplicações de Banco de Dados com InterBase 7.5 e MySQL 4.0.23
Érica

R$ 149,50

Introdução ao Desenvolvimento de Aplicações em Delphi
Uniderp
R$ 50,00

O Melhor da Delphi Line (CD 1 e 2)
Sistemas e Exemplos com fontes e Delphi
Ramos da Informática

10% desconto
De: R$ 134,00 Por: R$ 120,00

Delphi e Relatórios - Treinamento a Distancia (CD)2005
Clube Delphi
De: R$ 200,00 Por: R$ 180

Guia do Desenvolvedor de Delphi for .NET
Makron Books

10% desconto
De: R$ 173,50 Por: R$ 156,15

Delphi 8 Para Plataforma .NET: Curs Completo
Axcel
De: R$ 249,00 Por: R$ 224

Sistema Comercial Integrado com Delphi 2005: Cadastro e Estoque
Érica

10% desconto
De: R$ 135,00 Por: R$ 121,50

Frete Grátis
Para todo o Brasil!
3x no Visa e Master
e 6x no Hipercar

Outras facilidades de pagamento:

livrosdeprogramaca

Sendo assim, não foi preciso codificar nenhum comando SQL para persistir as alterações.

Fácil, não acha? Vamos ver na prática. Salve, compile e execute o projeto. Realize inclusões, alterações e exclusões à vontade, e em seguida clique no botão *Gravar Alterações*. Veja como mostra a **Figura 11**, que se todas as operações de persistência forem realizadas com sucesso, teremos uma mensagem de conclusão.

**Listagem 5. Persistindo as alterações no banco de dados**

```
try
  FbDataAdapter1.Update(dsCustomer);
  Label1.Text := 'Alterações persistidas no banco!';
except
  Label1.Text := 'Erro';
end;
```

Veja que estamos novamente utilizando uma estrutura *try...except* para tratamento de possíveis erros. E apenas chamando o método *Update* do *FbDataAdapter*, todas as alterações realizadas no *DataSet* serão persistidas no banco, onde basta passar o *DataSet* como parâmetro (temos outros parâmetros opcionais).

Você lembra que no início do artigo utilizamos um assistente para a configuração do *FbDataAdapter*? Foi naquele momento que os comandos *Insert*, *Update* e *Delete* foram criados e embutidos no método *Update* do *FbDataAdapter*.

## Conclusão

Pudemos ver como é simples e prático implementar funcionalidades de edição em um *DataGrid* do ASP.NET. Vimos também que, se utilizarmos o assistente de configuração do *FbDataAdapter*, pouparemos um bocado de código em nosso projeto. Como forma de aprimoramento do que foi visto, podemos customizar os controles usados para a edição dos campos. Como vimos, todos os campos foram apresentados como *TextBoxes*, mas poderíamos trazer campos relacionados, para que o usuário possa escolher o país em uma lista, por exemplo (campo *Country* da tabela *Customer*). Indico as vídeo aulas e artigos do Portal do Assinante, que mostram como adicionar um *DropDownList* e como fazer validações com os *Validators* diretamente em um *DataGrid*. Vimos também como é importante o uso dos *DataSets* em projetos com acesso a dados. Veremos em um próximo artigo como aproveitar ao máximo o uso dos *DataSet* entre outros recursos do ADO.NET. Até a próxima! ■

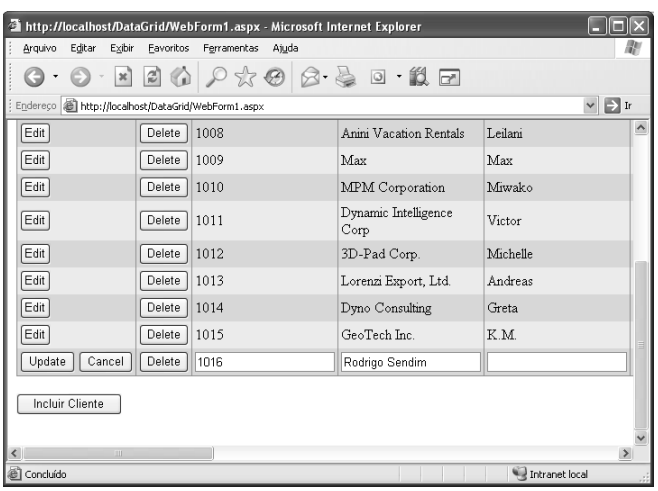



**Figura 10.** Incluindo um novo registro no DataGrid

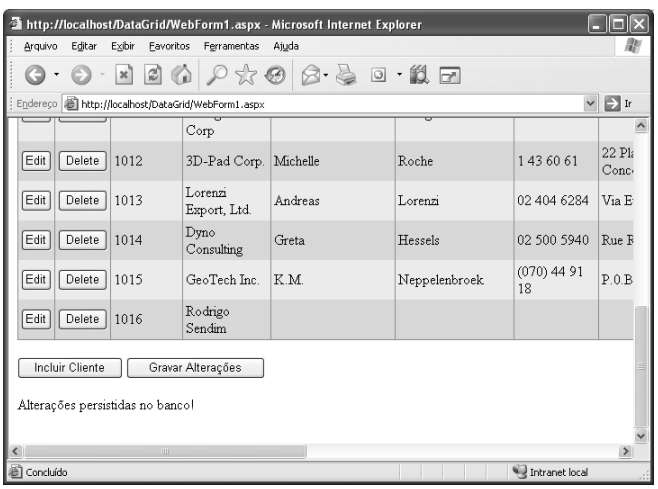
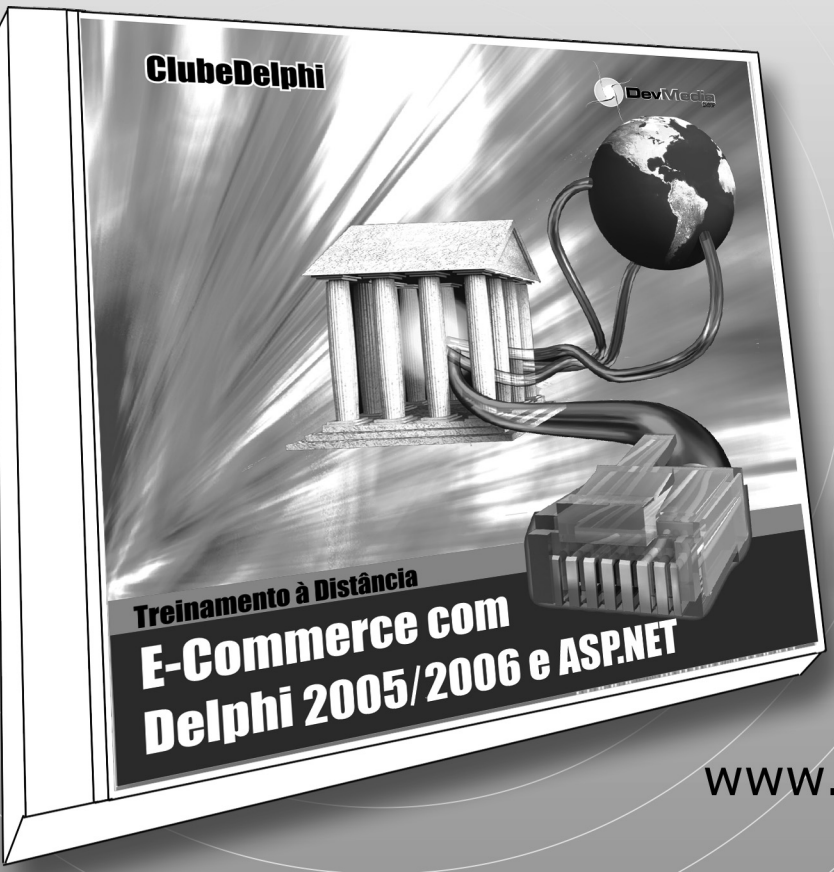



**Figura 11.** Persistindo os dados no banco

# Treinamentos a distância

## Aprenda de forma prática, eficiente e econômica.




### E-Commerce com Delphi 2005/2006 e ASP.NET

Aprenda a desenvolver poderosas aplicações Web através do Delphi 2005/2006 e as tecnologias ASP.N WebBroker, Intraweb e WebSnap.
O curso conta com módulos textuais;
35 horas de vídeo-aulas;
fórum exclusivo para os alunos e suporte on-line.

COMPRE JÁ O SEU
www.devmedia.com.br/curso

Outros Curso

### Delphi e Relatórios

Com este curso você desenvolverá relatórios através do Delphi 7, do Delphi 2005 e Delphi 2006, utilizando as ferramentas Rave Reports, Quick Reports e Crystal Report. Além do módulo textual o curso contém mais de 20 horas de vídeo-aulas; fórum exclusivo e suporte

### Evento à distância - Introdução ao .NET

Conheça os recursos do ASP.NET com o Visual Studio 2003 e aprenda a criar páginas ASP.NET; Webservices; trabalhar com a IDE do VS; componentização; DataGrid e criar aplicações de BD com o ADO.NET! São 6h de palestras em formato de vídeo-aulas e você pode tirar suas dúvidas com os palestrantes!

### Evento à distância - Novidades do Delphi 2005

Conheça os recursos do Delphi 2005 e aprenda a criar páginas ASP.NET; Webservices; trabalhar com a nova IDE do Delphi; utilizar programação OO no Delphi 2005; trabalhar em equipe com o novo Starteam e criar aplicações de BD com o ADO.NET!

DevMedia

# Mini-Curso de Delphi e UML – Parte III

**PAULO ROBERTO QUICOLI**
(pauloquicoli@gmail.com) é analista e programador da Control-M Informática. Trabalha com Delphi, desde sua primeira versão, e Firebird desenvolvendo aplicações cliente-servidor. Formado em Tecnologia de Processamento de dados pela FATEC, na cidade de Taquaritinga/SP


Dando continuidade ao curso, veremos neste artigo como gerar o código Delphi a partir do ModelMaker para construirmos nossa aplicação.

## Codificando e gerando classes

Inicie um novo projeto no Delphi e salve-o. Inclua uma nova unit e salve-a como "BusinessModel.pas". Acesse o menu *ModelMaker>Run ModelMaker Pascal Edition*. Agora no ModelMaker abra o projeto do curso e na guia *Modules* clique com o botão direito sobre *Classes not assigned to modules* e escolha *Add new Module*.

Agora é preciso escolher em qual unit as classes serão geradas e quais classes serão geradas nessa unit. É possível determinar que cada classe tenha sua própria unit ou gerar todas as classes em uma só. Veja na **Figura 1**, como deve ficar o módulo.

> **Nota:** Eu recomendo a geração de uma unit por classe, isso facilita na hora da manutenção, para dividir as tarefas para a equipe. Como nosso projeto é simples, decidi criar as classes em uma única unit, a BusinessModel.

Vamos implementar os métodos *Get/Set* das propriedades das classes *TRevista* e *TArtigo*, e faremos isso dentro do próprio ModelMaker. Na guia *Classes*, clique sobre *TRevista*, observe que abaixo são exibidas as propriedades, métodos e campos da classe, de acordo com o filtro de visualização. Clique sobre o método *GetEmail*, acesse a guia *Implementation* e digite o seguinte código:

```
Result := _Email.Value;
```

Implemente também o método *SetEmail* com o seguinte código:

```
_Email.Value := Value;
```

Faça o mesmo para os outros *Gets/Sets* da classe *TRevista*. Ao terminar vamos gerar a classe *TRevista*. Primeiramente devemos habilitar a geração de código, que por padrão é bloqueada. Observe que temos na barra de atalhos dois botões que são representados por um cadeado aberto e outro fechado (**Figura 2**).

Clique sobre o cadeado aberto, para a geração de código ficar habilitada. Na guia *Modules* clique sobre o botão *Generate* ( ), caso seja solicitada alguma confirmação, por hora, confirme. Acesse o Delphi, sem fechar o ModelMaker e veja que ambas as classes *TRevista* e *TArtigo* estão implementadas.

Você vai observar que os métodos de *TArtigo* não estão completos. Para mostrar o nível de integração do Delphi e do ModelMaker vamos fazer o inverso, vamos implementar os métodos de *TArtigo* no Delphi e pedir para que sejam atualizados no ModelMaker.

Implemente os *Gets/Sets* como feito em *TRevista*, a diferença estará para o *Get/Set* da propriedade *TArtigo.Revista*, que deverá ser implementada conforme o seguinte código:

```
function TArtigo.GetRevista: TRevista;
begin
  Result := _Revista.Value as TRevista;
end;

procedure TArtigo.SetRevista(Value: TRevista);
begin
  _Revista.Value := Value;
end;
```

É preciso também adicionar à cláusula *uses* a unit *InstantPersistence* e criar a seção *Initialization* para registrar as classes, que é um requisito do InstantObjects, conforme o seguinte código:

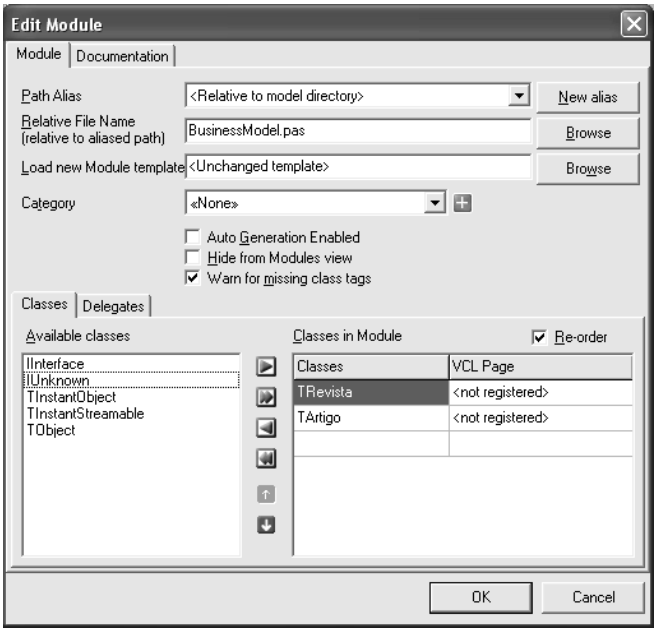



**Figura 1.** Escolhendo a unit onde serão geradas as classes

```
initialization
  InstantRegisterClasses([TRevista, TArtigo]);
```

Feito isso, acesse o menu *ModeMaker>Refresh in Model*, vá para o ModelMaker e veja as propriedades de *TArtigo* implementadas!

> **Nota:** sempre que fizer qualquer alteração no código do Delphi, utilize o menu anterior para atualizar o modelo do ModelMaker.

## Mapeamento das classes

Com as classes prontas, é preciso agora mapeá-las para um banco de dados relacional. Podemos fazer esse mapeamento através de *tags*, colocadas entre comentários, que o InstantObjects poderá interpretar. Veja na **Listagem 1** como fica o mapeamento para as classes.

**Listagem 1. Mapeamento de TRevista e TArtigo**

```
TRevista = class(TInstantObject)
    {IOMETADATA stored;
    Nome: String(50);
    Email: String(50);
    Link: string(50); }
  private
  ...
end;

TArtigo = class(TInstantObject)
  {IOMETADATA stored;
    Nome: String(70);
    NumeroEdicao: Integer;
    PalavrasChave: String(70);
    Resumo: Memo;
    Revista: Reference(TRevista); }
  private
  ...
end;
```

O InstantObjects oferece um assistente bem completo, que gera classes, *tags* de mapeamento e o banco de dados. Acesse o menu *View>InstantObjects Model Explorer*, para abrir a janela onde faremos o mapeamento.

Nessa janela, clique sobre o botão *Select Units* ( ), e localize a unit que contém as classes a serem mapeadas. Selecione então a unit *BusinessModel*, envie para a *Model Units* e confirme. O Model Explorer então exibirá as classes contidas na unit (**Figura 3**).

**Figura 2.** Habilitando a geração de código

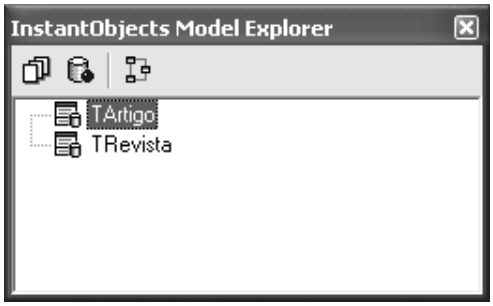



**Figura 3.** Model Explorer do InstantObjects

Clicando com o botão direito do mouse sobre as classes é aberto um menu por onde podemos editar nossa classe e configurar como suas propriedades serão exibidas na tela.

## Criando o banco de dados

No Model Explorer clique no botão *Build Database* ( ). Na janela *Database Builder* clique com o botão direito do mouse, escolha *New>IBX Connection* e mude seu nome para "conexao". É preciso configurar a nova conexão, clicando com o botão direito do mouse sobre ela e escolhendo *Edit*. Configure a conexão conforme a **Figura 4**, escolhendo um local para o seu banco de dados.

Clique em OK e no botão *Build*. Na janela de criação clique sobre *Show Build Sequence*, que mostrará a geração do banco. Finalmente, clique sobre *Build Database*, o InstantObjects notifica que se uma base de dados já existir ela perderá seus dados, e caso não seja essa a necessidade, utilize a opção *Evolve*. Como estamos criando um banco vazio, apenas confirme e o banco será criado.

> **Nota:** Se você estiver usando o Firebird e receber uma mensagem do tipo: *unavailable database*, normalmente basta adicionar o IP ou apenas *localhost* antes do caminho do banco na janela de configurações.

Adicione um DataModule ao projeto. Adicione um *IBDatabase* (paleta *InterBase*) e faça uma conexão com o banco que criamos anteriormente. Insira um *InstantIBXConnector* (*InstantObjects*) e ligue sua propriedade *Connection* ao *IBDatabase1*. Altere suas propriedades *IDDataType* para *dtString*, *IdSize* para "10", *IsDefault* para *True* e *LoginPrompt* para *False*.

Adicione também um *InstantSelector* e altere seu nome para "ArtigoSelector". O *InstantSelector* é responsável por realizar a seleção de objetos de uma determinada classe, essa que deve ser indicada na propriedade *ObjectClassName*, portanto configure-a para *TArtigo*.

Adicione um *DataSource* e ligue-o ao *ArtigoSelector*. Adicione mais um *InstantSelector*, dessa vez para a classe *TRevista*. Altere seu nome para "RevistaSelector" e *ObjectClassName* para *TRevista*. Adicione e ligue um *DataSource* ao *RevistaSelector*. Nesse momento, seu DataModule deve estar semelhante ao da **Figura 5**.

## Um pouco de Arquitetura

Muito se fala em "aplicações em camadas" e quando ouvimos isso logo pensamos em sistemas 3 camadas. Uma outra forma de desenvolvimento em camadas é organizar nossas classes logicamente, de acordo com o papel que exercem (**Figura 6**).

Os nomes e finalidades de cada camada podem variar de acordo com o entendimento, porém a idéia principal deve ser mantida: organizar as classes de acordo com sua função e sempre que possível manter a ordem de acessibilidade.

As camadas superiores fazem referência às camadas

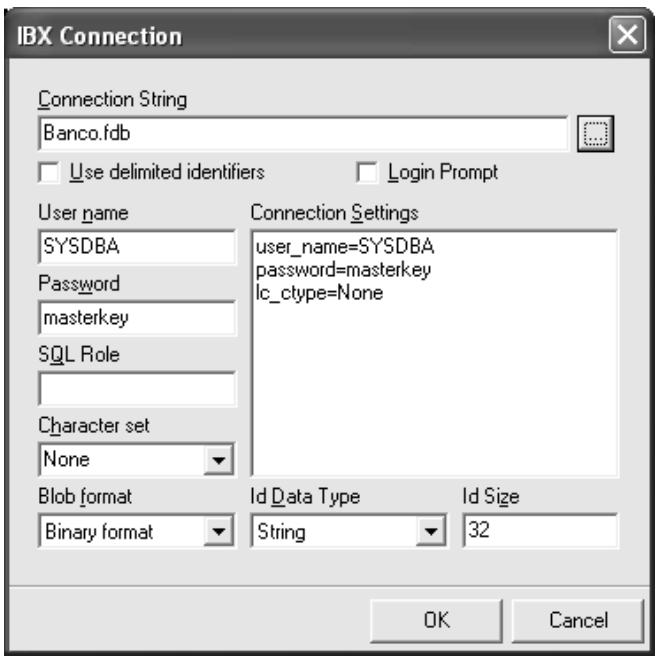



**Figura 4.** Configurando a conexão com o banco de dados

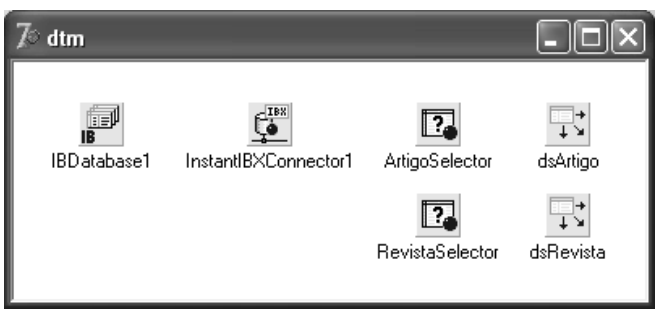



**Figura 5.** DataModule da aplicação

**Figura 6.** Arquitetura de camadas de sistemas

inferiores, levando sempre em conta que cada uma deve fazer o que foi projetada para fazer. No nosso exemplo, a camada de base de dados já foi implementada pelo InstantObjects, que é responsável em saber como acessar e lidar com os dados que serão mantidos em armazenamento físico.

Na camada de domínio, temos as classes *TRevista* e *TArtigo* que representam o negócio, porém estão muito pobres. Segundo Martin Fowler o design pobre em um domínio está no fato dele apenas conter as entidades que devem ser persistidas.

Por outro lado, um design rico de uma camada de negócios conduz a implementações de outras classes que auxiliam nas regras de negócio, como validações, consultas, listagem que posteriormente podem ser utilizadas pela camada de serviços. Vamos enriquecer então nosso domínio desenvolvendo uma classe de consulta, conforme solicitado no caso.

## Classe de consulta

No ModelMaker vamos adicionar um novo diagrama de classes e adicionar uma classe abstrata, chamada "TConsulta" e nela adicionar um método abstrato, chamado "Consultar", que recebe os seguintes parâmetros: "TipoConsulta: TTipo, Texto: string, Filtro: TRevista = nil". Você pode se perguntar agora, por que consulta é uma classe abstrata?

Existe uma boa prática, um princípio na programação OO dizendo que devemos programar para interfaces para obter modelos mais flexíveis. Vamos criar a classe de consulta que implementará o método *Consultar* de forma específica ao InstantObjects. Adicione uma classe que herda de *TConsulta* e sobreponha o método *Consultar* original (**Figura 7**).

Adicione as classes de consulta ao módulo *BusinessModel.pas* e gere o código, conforme fizemos anteriormente. Pelo Delphi vamos criar na unit *BusinessModel* um *Enum*, o *TTipo* que será utilizado na *Consulta*, conforme o seguinte código:

```
TTipo = (NomeArtigo, PalavraChave, Resumo);
```

Implemente o método *Consulta* conforme a **Listagem 2**.

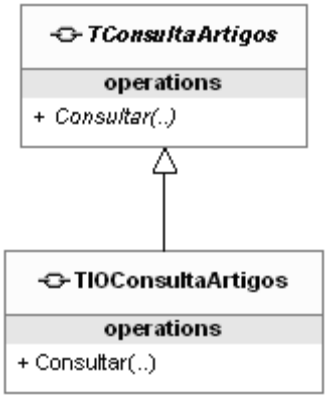



**Figura 7.** Diagrama de classes de Consultas

**Listagem 2. Método para consulta de artigos**

```
{ Adicione no uses a unit do DataModule }
procedure TIOConsultaArtigos.Consultar(
  TipoConsulta: TTipo; Texto: string;
  Filtro: TRevista = nil);
begin
   if Trim(Texto) = EmptyStr then
   begin
     raise Exception.Create(
       'Texto a ser buscado está vazio');
   end;
   case TipoConsulta of
     NomeArtigo:
       begin
         with dtm.ArtigoSelector do
         begin
           Close;
           Command.Clear;
           Command.Add('SELECT * FROM ANY TArtigo ');
           Command.Add('WHERE Nome LIKE ' +
             QuotedStr('%' + Texto));
           Command.Add('ORDER BY Nome');
         end;
       end;
     PalavraChave:
       begin
         with dtm.ArtigoSelector do
         begin
           Close;
           Command.Clear;
           Command.Add('SELECT * FROM ANY TArtigo ');
           Command.Add('WHERE PalavrasChave LIKE ' +
             QuotedStr('%' +Texto +'%'));
           Command.Add('ORDER BY Nome');
         end;
       end;
     resumo:
       begin
         with dtm.ArtigoSelector do
         begin
           Close;
           Command.Clear;
           Command.Add('SELECT * FROM ANY TArtigo ');
           Command.Add('WHERE Resumo LIKE ' +
             QuotedStr('%' +Texto+ '%'));
           Command.Add('ORDER BY Nome');
         end;
       end;
     end;
   if Filtro <> nil then
     dtm.ArtigoSelector.Command.Add(
       'AND Revista.Nome = ' + QuotedStr(
       Filtro.Nome));
   dtm.ArtigoSelector.Open;
   if dtm.ArtigoSelector.RecordCount = 0 then
     raise Exception.Create('Busca sem resultados');
end;
```

A primeira observação na implementação do método é como foi efetuada a busca. Veja que passamos para o *ArtigoSelector* um código que parece em muito o SQL padrão. Mas olhe bem, veja que não mencionamos nas sentenças nomes de tabela ou campos, mas nome de classes e suas propriedades.

Se dermos agora uma olhada na estrutura como foi montado o método, podemos ver que segue uma das implementações do padrão *Strategy*. No formulário principal da aplicação para utilizar nossa classe de consulta, adicione alguns componentes para que fique semelhante à **Figura 8**.

O *dsArtigo* deve estar ligado ao *dtm.ArtigoSelector* e o *DBGrid* ao *dsArtigo*. O *dsRevista* é ligado ao *RevistaSelector* (se preferir coloque o componente no DataModule), que possui as seguintes propriedades alteradas: *AutoOpen* para *True*, *ObjectClassName* para *TRevista*, *Connector* para *dtm.InstantIBXConnector1* e *Command* para "SELECT * FROM TRevista".

O *DBLookupComboBox1* mostrará o nome da revista utilizando o *dsRevista*. Codifique os eventos *OnShow* e *OnClose* do formulário com o seguinte código, respectivamente (não esqueça de adicionar a unit do DataModule):

```
OnShow
dtm.ArtigoSelector.Connector.Connect;

OnClose
dtm.ArtigoSelector.Connector.Disconnect;
```

Ao iniciar a aplicação conectamos no banco, ao fecharmos, desconectamos.

## Adicionando artigos e revistas

Vamos agora criar uma classe que pode ser identificada como uma classe de serviço, que é responsável por cadastrar ou editar um artigo. Antes, vamos definir os formulários que serão utilizados por essa classe. Teremos uma hierarquia simples de formulários, um formulário base de cadastro e mais dois especializados para *TArtigo* e *TRevista*. Crie um novo formulário e o defina conforme a **Figura 9**.

Crie um novo formulário, herdando-o de *TCadBaseF* e adicione um *InstantExposer*, da guia *InstantObjects*. Em *Name* digite "ArtigoExposer", em *ObjectClassName* escolha *TArtigo* e por fim em *FieldOptions.foObjects* escolha *True*. Adicione um *DataSource* ao formulário, chamando-o de "dsArtigo" e faça a ligação ao *ArtigoExposer*.

Dando um duplo clique sobre o *ArtigoExposer*, adicione as propriedades da classe e arraste-as para o formulário, onde automaticamente serão criados os campos e seus rótulos. Não adicione a propriedade *Revista.Nome*, pois nela vamos criar um campo *lookup*.

Insira um *InstantSelector* ajustando suas propriedades conforme a **Tabela 1**, ligando-o a um *DataSource* ("dsSelector").

| Propriedade | Valor |
|---|---|
| Name | RevistaSelector |
| AutoOpen | True |
| Command | SELECT * FROM TRevista |
| FieldOptions.foObjects | True |
| Connector | dtm.InstantIBXConnector1 |
| ObjectClassName | TRevista |

**Tabela 1.** Propriedades de dsSelector

Para o *lookup* utilize um *DBLookupComboBox*. Veja como configurá-lo na **Tabela 2**.

| Propriedade | Valor |
|---|---|
| DataSource | dsArtigo |
| DataField | Revista |
| ListSource | dsSelector |
| KeyField | Self |
| ListField | Nome |

**Tabela 2.** Propriedades do LookupComboBox

Organize a tela de cadastro de artigos conforme a **Figura 10**.

No evento *OnClick* dos botões *Salvar* e *Cancelar* digite o código da **Listagem 3**, ele respectivamente salva os dados inseridos e cancela a edição.

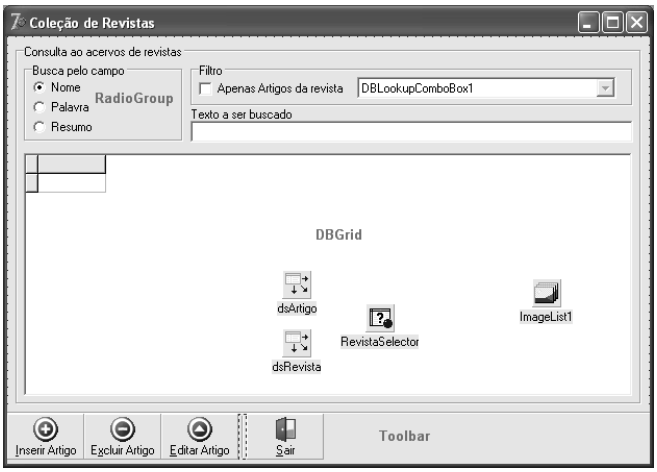



**Figura 8.** Tela principal do sistema de exemplo

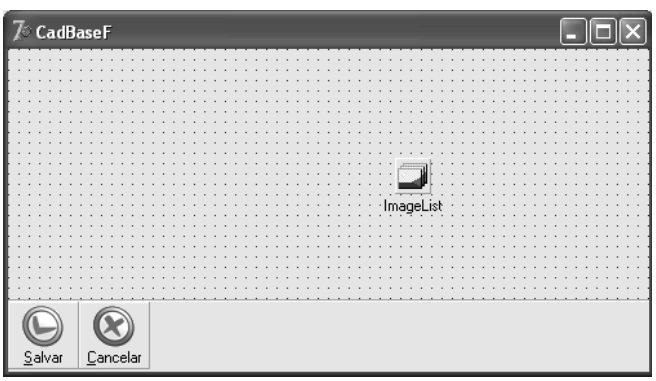



**Figura 9.** Formulário base para cadastros

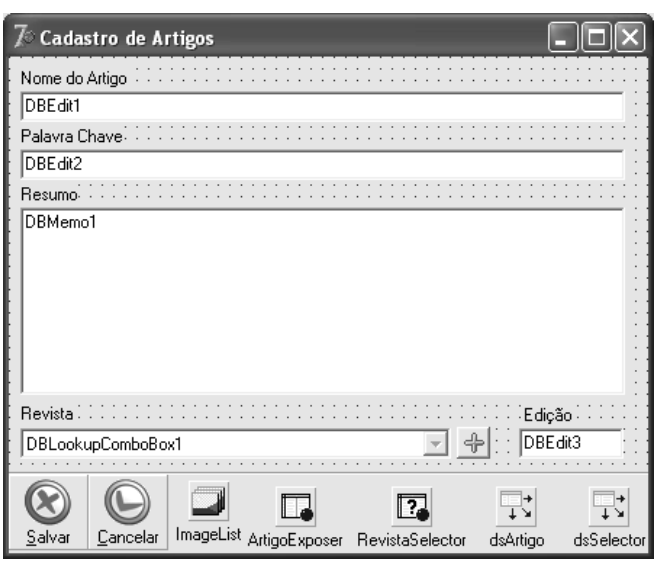



**Figura 10.** Tela para cadastro de artigos

**Listagem 3. Evento OnClick dos botões Salvar e Cancelar**

```
procedure TCadArtigoF.ToolButton1Click(
  Sender: TObject);
begin
  inherited;
  ArtigoExposer.PostChanges;
  ModalResult := mrOk;
end;

procedure TCadArtigoF.ToolButton2Click(
  Sender: TObject);
begin
  inherited;
  ModalResult := mrCancel;
end;
```

É preciso criar o cadastro das Revistas, assim crie mais um formulário que descenda de *TCadBase*. Adicione um *InstantExposer*, chame-o de "RevistaExposer", configurando-o como fizemos no cadastro de artigos, e o evento *OnClick* dos botões é similar também ao cadastro de artigos, com a diferença que referenciamos o *RevistaExposer*. Formate o formulário como na **Figura 11**.

Chegou a hora de implementar a classe de serviço. Adicione uma nova unit, salve-a como "Servicos.pas" e em sua cláusula *uses* faça referência à *BusinessModel.pas*. No ModelMaker, crie um novo diagrama de classes e dê o nome de "Classes – Serviço". Nesse diagrama insira uma classe chamada "TEdicao", com dois métodos estáticos:

```
EditRevista(Revista: TRevista): boolean
EditArtigo(Artigo: TArtigo): boolean
```

Adicione ao modelo a unit e mande o ModelMaker gerar o código da classe nessa unit e implemente os métodos como na **Listagem 4.**

**Listagem 4. Classe de serviço da aplicação**

```
implementation

...

{ Adicione no uses a unit BusinessModel e a unit de
  cada formulário de cadastro }
class function TEdicao.EditArtigo(
  Artigo: TArtigo): Boolean;
var
  Form: TCadArtigoF;
begin
  Form := TCadArtigoF.Create(nil);
  try
    Form.ArtigoExposer.Subject := Artigo;
    Result := Form.ShowModal = mrOk;
  finally
    Form.Free;
  end;
end;

class function TEdicao.EditRevista(
  Revista: TRevista): Boolean;
var
  Form: TCadRevistaF;
begin
  Form := TCadRevistaF.Create(nil);
  try
    Form.RevistaExposer.Subject := Revista;
    Result := Form.ShowModal = mrOk;
  finally
    Form.Free;
  end;
end;
```

## Fazendo "rodar"

Para fazer a consulta criamos a classe *TIOConsultaArtigos*, portanto vamos usá-la no evento *OnKeyUp* do campo de texto a ser buscado (no formulário principal), conforme a **Listagem 5** (atente para os nomes dos componentes).

**Figura 11.** Tela de cadastro de revistas

**Listagem 5. Realizando a busca**

```
procedure TPrincF.EditTextoKeyUp(Sender: TObject;
  var Key: Word; Shift: TShiftState);
var
 lConsulta: TIOConsultaArtigos;
 lRevista: TRevista;
begin
  lRevista := nil;
  if ckFiltro.Checked then
    lRevista :=
      dtm.RevistaSelector.CurrentObject as TRevista;
  if Key = VK_Return then
  begin
    lConsulta := TIOConsultaArtigos.Create;
    case RadioGroup1.ItemIndex of
      0: lConsulta.Consultar(NomeArtigo,
        EditTexto.Text, lRevista);
      1: lConsulta.Consultar(PalavraChave,
        EditTexto.Text, lRevista);
      2: lConsulta.Consultar(Resumo, EditTexto.Text,
      lRevista);
    end;
    lConsulta.Free;
  end;
end;
```

Para cadastrar um artigo, no botão *Inserir Artigo* temos o código da **Listagem 6**.

**Listagem 6. Inserindo um novo artigo**

```
{ Declare no uses a unit da classe TEdicao }
procedure TPrincF.btInserirClick(Sender: TObject);
var
  lArtigo: TArtigo;
begin
  lArtigo := TArtigo.Create;
  try
    if TEdicao.EditArtigo(lArtigo) then
      dtm.ArtigoSelector.AddObject(lArtigo);
  finally
    lArtigo.Free;
  end;
end;
```

Para excluir, chamamos o método *Delete* do *DataSet*:

```
dtm.dsArtigo.DataSet.Delete;
```

Para editar, precisamos obter qual objeto *TArtigo* está selecionado e após sua edição atualizamos sua exibição:

```
if TEdicao.EditArtigo(
  dtm.ArtigoSelector.CurrentObject as TArtigo) then
  dtm.ArtigoSelector.Refresh;
```

Veja que nos códigos anteriores fazemos uso da classe de serviço. No formulário principal, não fazemos menção alguma aos formulários de cadastro, apenas a classe *TEdicao* sabe como utilizá-los. Isso chama-se *encapsulamento*.

Não criamos um acesso direto ao cadastro de Revistas, sendo feito através do cadastro de Artigos. Veja que ao lado do *DBLookupComboBox* temos um *SpeedButton*, que é encarregado de chamar o cadastro de Revistas. Veja sua implementação na **Listagem 7**.

**Listagem 7. Abrindo o formulário de cadastro de revistas**

```
{ Adicione no uses BusinessModel e Serviços }
procedure TCadArtigoF.btAdicionaRevistaClick(
  Sender: TObject);
var
  lRevista: TRevista;
begin
  inherited;
  lRevista := TRevista.Create;
  if TEdicao.EditRevista(lRevista) then
    RevistaSelector.AddObject(lRevista);
  lRevista.Free;
  ComboRevistas.Refresh;
end;
```

Após tudo isso, temos o aplicativo pronto, mesmo que simples, em camadas (**Figura 12**). Na **Figura 13** temos a aplicação em execução.

## Conclusão

Ao longo desses artigos, vimos que o desenvolvimento Orientado a Objetos não é moroso como diz a lenda. Em minha opinião tal afirmação é utilizada por aqueles que não conhecem OO e têm medo de mudanças, mesmo que positivas. Utilizando a documentação de Caso de Uso, o desenvolvimento torna-se mais dirigido a atender o que o usuário realmente precisa e solicita.

Utilizando ferramentas eficientes como Delphi, ModelMaker e o framework InstantObjects ganhamos tempo porque mantemos nosso foco nas regras de negócio e nos problemas do nosso cliente (e não de infra-estrutura). Um abraço a todos e até a próxima! ■

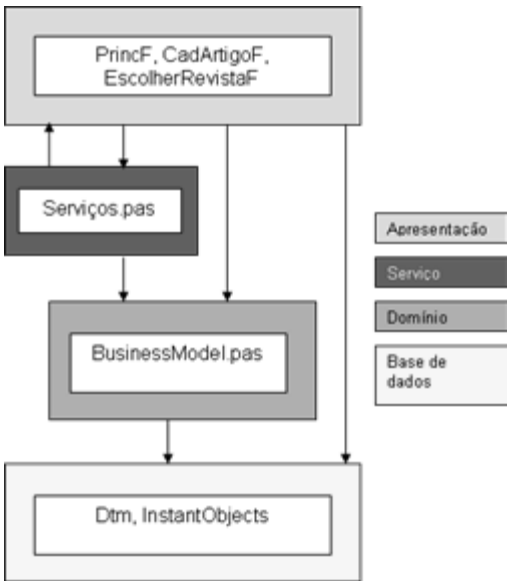



**Figura 12.** Camadas do sistema de exemplo

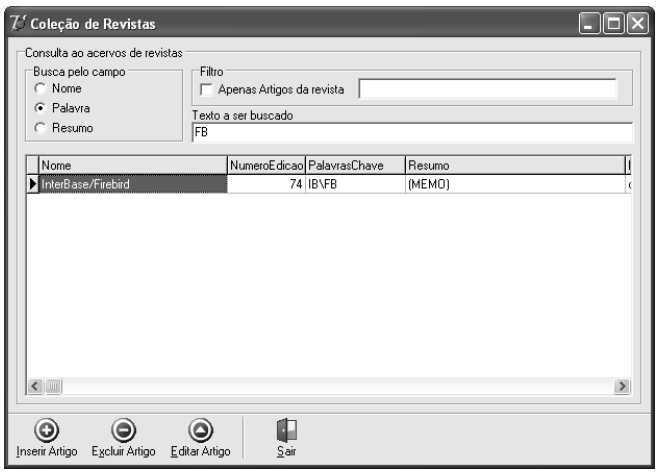
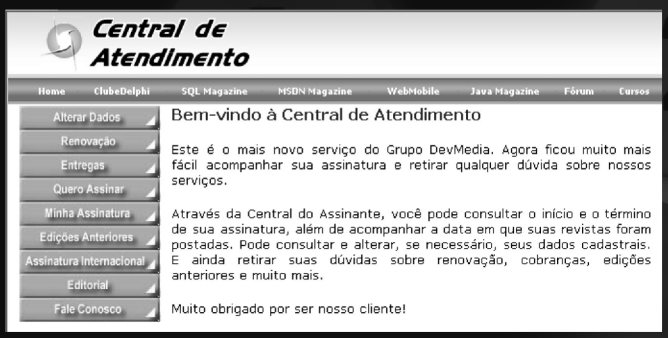



**Figura 13.** Aplicação sendo executada

# Fale conosco!




Conheça a Central de Atendimento on-line da DevMedia. Agora ficou muito mais fácil de acompanhar sua assinatura, alterar seus dados cadastrais e retirar qualquer dúvida sobre nossos serviços e produtos. Para isso basta ter em mãos seu login e senha.

| Dados Cadastrais | |
|---|---|
| Login: | Ricsilva |
| Senha: | rsilva2006 |
| Nome: | Ricardo da Silva |
| Empresa: | Minha Empresa |
| E-Mail: | ricsilva@email.com |
| E-Mail Trab: | workgroup@email.com |
| Endereço: | Rua das Marrecas, 32 |
| Bairro: | Centro |
| Cidade: | Rio de Janeiro |
| Estado: | RJ |
| País: | Brasil |
| Cep: | 12327-500 |
| Tel. Comercial: | 21 2654-8697 |
| Tel. Residêncial: | 21 3879-6123 |
| Tel. Contato: | 21 9854-2348 |

Editar Cadastro

| Assinaturas Ativas |
|---|
| ClubeDelphi - Revistas Enviadas |
| MSDN Magazine - Revistas Enviadas |

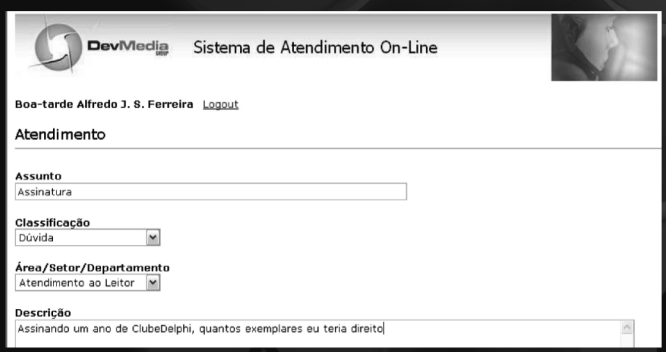
**Na Central de Atendimento você:**

**- Consulta a data de envio de cada exemplar**

**- Consulta as perguntas mais frequentes**

**- Renova por boleto ou cartão de crédito**

**- Consulta e altera seus dados cadastrais**




Para entrar em contato com a DevMedia, basta postar sua dúvida na própria Central de Atendimento através de um sistema web, prático, seguro e fácil de usar. Com essa nova ferramenta você tem a garantia de receber suas respostas em menor tempo e com alto padrão de qualidade.

**www.devmedia.com.br/central**

DevMedia

# VCL Revelada

**MICHAEL BENFORD**
(mrbenford@gmail.com)
é Acadêmico do curso de matemática da Universidade Federal Fluminense e desenvolvedor Delphi e .NET da GPS Tecnologia. Programa em Delphi há 5 anos, desenvolvendo aplicações cliente/servidor, multicamadas, Web, utilitários de uso geral, componentes, experts e aplicativos não-comerciais. É colunista do portal da DevMedia, articulista e membro da comissão editorial da revista ClubeDelphi.

Nossa ferramenta de desenvolvimento já tem mais de 10 anos de vida. Quantos programas foram feitos nesse período! Muitos deles, senão todos, foram feitos com o auxílio de componentes, controles, objetos etc., sejam para apenas exibir um botão na tela ou para fornecer acesso a um banco de dados.

E, independentemente da função dos componentes, a sua utilização (e criação) foi enormemente facilitada através de uma sigla que conhecemos bem: VCL. Mas, como a VCL funciona? Quais classes a sustentam? Entender o seu funcionamento interno é de suma importância para criar novos componentes corretamente, e saber como se comportam os que são nativos do Delphi. Isso tudo veremos neste artigo.

## VCL: um pequeno histórico

Em inglês, VCL significa *Visual Component Library*. Em português, *Biblioteca de Componentes Visuais*. Embora o nome faça parecer que a VCL é composta apenas de controles visíveis ao usuário, também há classes que, mesmo não aparecendo no sentido literal do termo, servem como base ou auxiliam as que assim o são.

A VCL existe desde o Delphi 1, lançado em 1995, e assim como a ferramenta, vem sofrendo mudanças e melhorias ao longo do tempo. Sua principal função era abstrair do desenvolvedor o conhecimento de API's nativas do sistema operacional para a criação de interfaces gráficas (*GUI* ou *Graphical User Interface*).

Esse papel, todos sabemos que a VCL cumpre muito bem, e isso foi um dos grandes atrativos para o Delphi, quando do seu lançamento. Afinal, alguém aqui já precisou escrever *CreateWindow* para criar um formulário?

## Hierarquia da VCL

A hierarquia da VCL é absurdamente extensa. Nela es-

tão todos os componentes nativos do Delphi/C++ Builder. Nesse grupo de classes, podemos destacar um subconjunto que serve de base para todos os outros:

- TObject
  - TPersistent
    - TComponent
- TControl
  - TGraphicControl
  - TWinControl
    - TCustomControl

Veremos agora a função, alguns métodos e propriedades de cada classe.

## TObject

Essa classe é a base para todos os objetos na Delphi Language. Todas as classes da VCL herdam dela, direta ou indiretamente. *TObject* provê as funções básicas de todo objeto, como criação, destruição, alocação e liberação de memória. Além disso, também publica informações de *run-time* (*RTTI* ou *Runtime Type Information*) sobre as propriedades declaradas em sua seção *published*.

Essa classe não possui propriedades nem eventos, apenas métodos, muitos estáticos (também chamados de métodos de classe). Algumas dessas rotinas são bem interessantes em determinadas situações. Vejamos então quais são elas:

*AfterConstruction*: chamado após o construtor da classe ter sido executado com sucesso. Na implementação original em *TObject* ele não faz nada. Você deve, portanto codificá-lo em uma classe derivada. *TCustomForm*, por exemplo, utiliza esse método para gerar o evento *OnCreate*. Como essa rotina é chamada automaticamente após o construtor da classe, você nunca deve chamá-la explicitamente;

*BeforeDestruction*: sua função é a oposta do método anterior. É chamado imediatamente antes do destrutor da classe ser executado. Novamente, não chame essa rotina diretamente, pois ela é disparada automaticamente em momento oportuno. É importante notar que o método só será chamado se o construtor da classe for bem sucedido. Isso quer dizer que se uma exceção ocorrer dentro do construtor, o destrutor será chamado para liberar a memória do objeto alocada até o momento do erro, mas *BeforeDestruction* não será executado;

*ClassName*: retorna o nome da classe do objeto. Por exemplo, um botão retornaria *TButton*, um *Label* retornaria *TLabel* etc. É muito útil quando precisamos diferenciar instâncias de objetos armazenadas em variáveis cujo tipo seja uma classe ancestral;

*ClassNameIs*: determina se o objeto é do tipo informado. Como o item anterior, é útil quando se faz criação dinâmica de objetos;

*ClassParent*: retorna a classe ancestral do objeto. Pode ser usado, por exemplo, para verificar a "hereditariedade" de uma instância. Para uma aplicação prática do que vimos até aqui, inicie uma nova aplicação e coloque um *Button* e um *ListBox* no formulário. Em seguida implemente o evento *OnClick* do botão para com o código da **Listagem 1**.

**Listagem 1. ClassName, ClassType e ClassParent**

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ClassRef: TClass;
begin
  ListBox1.Clear;
  ClassRef := Sender.ClassType;
  while ClassRef <> nil do
  begin
    ListBox1.Items.Add(ClassRef.ClassName);
    ClassRef := ClassRef.ClassParent;
  end;
end;
```

Após executar a aplicação e clicar no botão, você receberá uma lista como a seguinte, que mostrará toda a hierarquia de classes a partir de *TButton*:

- TButton
- TButtonControl
- TWinControl
- TControl
- TComponent
- TPersistent
- TObject

> Dica: Quando possível, evite usar ClassParent, dando preferência aos operadores is e as da Delphi Language.

*ClassType*: Retorna o tipo da classe do objeto. Geralmente é usado em conjunto com o método anterior. Na **Listagem 1** você pode ver essa função sendo utilizada. A mesma observação feita antes vale aqui: na maioria dos casos use *is* e *as*;

*Free*: conhecido por grande parte dos desenvolvedores Delphi. Sua função é liberar a memória alocada pelo objeto, se necessária. A vantagem de usar *Free* no lugar do destrutor da classe é o fato de ser realizada uma verificação de nulidade. Assim ele pode ser chamado mesmo que a instância do objeto seja *nil*, sem que um erro de violação de acesso seja disparado;

> **Dica:** Consulte na edição 71 da revista ClubeDelphi o artigo Criação dinâmica de componentes, para ver em mais detalhes como usar corretamente o Free.

*InheritsFrom*: verifica se o objeto em questão descende ou não de uma determinada classe. Algo similar pode ser feito com o operador *is*, mas apenas em instâncias de objetos.

Note que, em sua maioria, esses métodos são estáticos, como *ClassParent* e *InheritsFrom*, e podem ser chamados independentemente de haver uma instância do objeto. É exatamente isso que não os deixa se tornarem obsoletos.

Como o leitor pode notar, *TObject* oferece métodos pouco úteis, com exceção de *Free*, para o nosso dia a dia. Tais funções se tornam interessantes e necessárias quando es-

tamos desenvolvendo aplicativos que dependam de informações dinâmicas, com RTTI.

## TPersistent

*Persistência* é uma coisa que usamos frequentemente quando trabalhamos com componentes. Mas, o que significa o termo nesse contexto? Veja um exemplo: um botão é colocado em um formulário, e sua propriedade *Caption* é alterada para "Olá mundo". Em seguida o formulário é salvo e fechado. Ao carregá-lo novamente, qual será o valor da propriedade que modificamos anteriormente? A resposta é óbvia: "Olá mundo".

Você, leitor, deve estar pensando: "Mas o que isso tem a ver com *TPersistent*?". Tudo! Por padrão, o Delphi em si, não sabe como as propriedades de componentes devem ser salvas. Para isso, é necessário que cada classe diga ao IDE como o fazer.

Pois bem, a classe que introduz os métodos, que devem ser implementados, para permitir o salvamento e recuperação dessas propriedades é, como agora você já deva ter descoberto, *TPersistent*. De fato, todos os componentes herdam indiretamente dessa classe e, portanto, sabem como persistir seus atributos.

> **Nota:** Essa persistência é feita gravando os dados da memória para o arquivo DFM (ou XFM, no caso de aplicações CLX) e vice-versa.

> **Nota:** Nunca instancie diretamente TPersistent. Sempre crie classes que herdem dela para poder criar objetos.

Existem classes que também podem herdar de *TPesistent*, não sendo necessariamente componentes, mas que precisem salvar suas propriedades no arquivo do formulário. Tais objetos, em função disso, podem aparecer em *design time*, não de forma independente, mas sim implementando propriedades de outros componentes. É o caso das classes *TBrush*, *TBitmap*, *TFont*, *TStringList*, *TCollection* etc. Nessa situação são comumente conhecidas como subpropriedades de um componente.

Além de fornecer o mecanismo de persistência, *TPersistent* também dá suporte à atribuição de objetos para outros. Para entender melhor, veja o código da **Listagem 2**.

**Listagem 2. Atribuição de ponteiros ou propriedades?**

```
var
  { TMeuObjeto herda de TPersistent }
  Objeto1, Objeto2: TMeuObjeto;
begin
  Objeto1 := TMeuObjeto.Create;
  Objeto2 := TMeuObjeto.Create;

  Objeto1.Propriedade1 := 'Teste';
  Objeto1.Propriedade2 := 10;

  Objeto2 := Objeto1;
end;
```

Como o leitor já sabe, o código da listagem anterior não copiará os valores das propriedades de Objeto1 em Objeto2, mas sim faz ambos os objetos apontarem para a mesma instância em memória. A cópia desejada de valores pode ser obtida através do método *Assign*, disponibilizado virtualmente por *TPersistent*, desde que devidamente implementado pelas classes descendentes. Então, o código anterior poderia ser reescrito para:

```
Objeto2.Assign(Objeto1);
```

Como podemos perceber até aqui, *TPersistent* é uma das mais importantes classes da VCL. Mas ela não trabalha sozinha e veremos sua influência no próximo tópico.

## TComponent

Essa classe é o coração de todo componente no Delphi. Eles herdam direta (componentes não-visuais) ou indiretamente (componentes visuais) dela. Na hierarquia da VCL, *TComponent* vem logo abaixo de *TPersistent*, e logo implementa métodos para a persistência de suas propriedades publicadas. O mecanismo de persistência é especial para essa classe: as propriedades declaradas na seção *published* são automaticamente gravadas e restauradas com o arquivo do formulário, como pudemos ver no exemplo no início do tópico anterior.

Além dessas características, *TComponent* também introduz o conceito de **proprietário** (*owner*) entre componentes. Um componente pode ser o dono de outros, e ao mesmo tempo ser de propriedade de mais outros. Essa relação é estabelecida através do parâmetro *AOwner* do construtor da classe:

```
constructor TComponent.Create(AOwner: TComponent);
```

O proprietário é responsável por duas coisas: carregar e salvar as propriedades publicadas dos seus "filhos"; e liberá-los da memória quando for destruído.

> **Nota**: Se um componente tem como Owner outro que não seja um formulário ou um DataModule, suas propriedades não serão salvas e carregadas junto com ele. Nesse caso é necessário configurar o componente como um sub-componente. Isso é feito através do método SetSubComponent, visto adiante.

Existem duas propriedades oferecidas por *TComponent* para trabalhar com essa relação entre componentes:

- Owner: indica qual componente é o proprietário;
- Components: fornece a coleção de componentes "possuídos" por esse.

Essas propriedades são muito úteis quando estamos de-

senvolvendo aplicações onde os componentes são criados em *runtime*.

A classe *TComponent* também oferece alguns métodos que podem ser bem úteis. Vejamos alguns deles:

*FindComponent*: retorna a instância de um objeto através do seu nome. Essencial em aplicações que utilizem criação dinâmica de componentes;

*InsertComponent*: troca o *owner* de um componente pelo chamador do método. Comumente é usado em conjunto com o método *RemoveComponent*. Veja na **Listagem 3** um exemplo de ambos.

**Listagem 3. Exemplo de InsertComponent e RemoveComponent**

```
var
  I: Integer;
  Temp: TComponent;
begin
  for I := ComponentCount - 1 downto 0 do
  begin
    Temp := Components[I];
    if not (Temp is TControl) then
    begin
      RemoveComponent(Temp);
      DataModule.InsertComponent(Temp);
    end;
  end;
end;
```

> **Nota:** Esses métodos não modificam os arquivos dos formulários (ou DataModules) envolvidos. Ou seja, as mudanças não são persistidas.

*IsImplementorOf*: indica se o componente implementa a interface informada. É similar, até certo ponto, à função *Supports*, e em algumas situações pode tornar a leitura do código mais legível;

*Loaded*: inicializa o componente após sua leitura do arquivo do formulário. Você não deve chamar esse método manualmente, pois isso é feito pelo Delphi. Porém, pode sobrescrevê-lo para inserir código que deve ser executado imediatamente após o carregamento do componente para a memória. Muito útil quando em desenvolvimento de componentes;

*PaletteCreated*: chamado quando o componente é criado através da paleta de componentes do Delphi. Você, portanto, não deve chamá-lo explicitamente, mas sim sobrescrevê-lo, como com *Loaded*, para ser notificado da criação do componente pela *Component Palette* do IDE;

*RemoveComponent*: exclui o componente informado da lista de componentes do chamador. Como já vimos, pode ser usado em conjunto com *InsertComponent* (**Listagem 3**).

> **Nota:** Tenha cuidado ao usar esse método. Antes de excluir um componente do seu owner, guarde uma referência para ele, para ser possível o seu posterior acesso.

*SetSubComponent*: define o componente com um sub-componente, que é um componente cujo *owner* não é nem um formulário nem um DataModule, permitindo que suas propriedades publicadas possam ser salvas automaticamente no arquivo do formulário ou DataModule em questão.

Por fim, não podemos esquecer que é em *TComponent* que a propriedade *Tag* aparece inicialmente. Essa propriedade como muitos sabem, não tem significado específico, podendo ser utilizada livremente pelo desenvolvedor para armazenar valores juntamente com o componente. Porém, a maioria a utiliza apenas para guardar valores inteiros, mas é possível guardar qualquer estrutura dentro dela, como mostrado na **Listagem 4**.

**Listagem 4. Utilizando Tag para armazenar um record**

```
type
  TCliente = record
    Nome: string;
    Idade: Integer;
    Adimplente: Boolean;
  end;

var
  Cliente: TCliente;
...
procedure SetTag;
begin
  Cliente.Nome := 'Michael Benford';
  Cliente.Idade := 21;
  Cliente.Adimplente := True;

  Button.Tag := Integer(@Cliente);
end;

procedure GetTag;
var
  PCliente: ^TCliente;
begin
  PCliente := Pointer(Button.Tag);
  with PCliente^ do
    ShowMessage(Nome + sLineBreak + IntToStr(Idade) +
      sLineBreak + BoolToStr(Adimplente));
end;
```

Como todo ponteiro pode ser representado por um inteiro, é fácil guardar qualquer tipo de dado em uma *Tag*. Apenas tome o cuidado de não liberar a memória representada pelo ponteiro armazenado sem atualizar a propriedade, pois caso contrário ocorrerá um erro de violação de acesso no momento em que se tentar acessar a área da memória que não existe mais.

## TControl

Na VCL, os componentes são divididos em dois grandes grupos: visuais e não-visuais. Entende-se por componente visual aquele que será visível em *runtime* e irá interagir com o usuário de alguma forma. No jargão da VCL, chama-se de **controle** o componente que pertencer a esse conjunto, e apenas de **componente** o que for englobado pelo outro grupo.

> **Nota**: De agora em diante vamos usar esses termos para referenciar corretamente os componentes. Como vimos no tópico anterior, os componentes são derivados de TComponent. Já os controles, como você já deve supor, derivam de TControl.

Nessa classe são primeiramente introduzidos métodos, propriedades e eventos que gerenciam a aparência e o comportamento do controle. É o caso das propriedades *Caption*, *Left*, *Top*, *Height* e *Width*, dos métodos *Click*, *Show*

e *Hide*; e dos eventos *OnClick*, *OnMouseMouse* e *OnResize*, e muitos outros. Embora herdados por todos os controles visuais, tais atributos são publicados de acordo com a função de cada classe. Por exemplo, o *Image* não publica a propriedades *Caption* e *Font*, nem o evento *OnResize*.

Outro aspecto importante que *TControl* introduz é o conceito de container visual, obtido através da propriedade *Parent*. Esse atributo indica qual controle será o "pai" desse, isso é, qual controle que irá contê-lo visualmente.

Algo essencial que deve ser falado sobre controles, derivados diretamente de *TControl*, é o fato de não receberem a entrada do teclado. Dessa forma, não há o conceito de foco, e a única interação possível com o usuário é através do mouse.

A maioria dos métodos, eventos e propriedades dessa classe já são largamente conhecidos pelos desenvolvedores Delphi. Porém, vale a pena destacar alguns não tão "populares":

*ControlState*: informa o estado atual do controle. Seu uso é maior entre criadores de componentes, mas há alguns estados oferecidos que podem ser utilizados também em aplicativos. Como exemplo, inicie uma nova aplicação VCL e coloque um *Button* sobre o formulário. Em seguida implemente o seu evento *OnMouseMove* como mostrado na **Listagem 5**, que permite arrastar um controle com auxílio de *ControlState*.

**Listagem 5. Arrastando o controle com o auxílio de ControlState**

```
procedure TForm1.Button1MouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  with Sender as TControl do
  if csLButtonDown in ControlState then
  begin
    Left := ClientToParent(
      Point(X, Y)).X - (Width div 2);
    Top  := ClientToParent(
      Point(X, Y)).Y - (Height div 2);
  end;
end;
```

> **Nota:** Foi feito o typecast de Sender para TControl apenas para frisar que as propriedades utilizadas no código aparecem inicialmente neste classe.

*ControlState* é um *set* e possui um elemento chamado *csLButtonDown*, que informa se o botão esquerdo do mouse está pressionado sobre o controle. Se estiver, então movemos o botão pelo formulário com o ponteiro do mouse. Alguns, em situações semelhante, podem ter usado um *flag* auxiliar para saber se o botão do mouse estava pressionado. Observe como a implementação fica bem mais simples como mostrado.

*ControlStyle*: são definidas várias características do estilo do controle. É possível, por exemplo, impedir que o controle receba os cliques do mouse, ou fixar sua altura e largura, ou ainda saber se existe uma ação associada a ele. Não deve ser modificada em *runtime*, exceto dentro de um construtor de classe. Ou seja, seu uso é restrito ao código de criação de um novo controle;

*ClientHeight* e *ClientWidth*: respectivamente, a altura e a largura útil do controle. Como implementado em *TControl*, possuem os mesmos valores de *Height* e *Width*, nessa ordem. Em outras classes, o seu significado pode mudar, como em *TForm*, onde *ClientHeight* é a medida da altura do formulário menos a altura da barra de título da janela;

*BringToFront* e *SendToBack*: enviam o controle para frente e para trás, respectivamente, dos demais controles dentro do mesmo container, mudando a chamada **ordem Z** (*z order*). É importante notar que controles derivados de *TGraphicControl* e *TWinControl*, vistos mais adiante, sempre ficarão dispostos à frente de controles descendentes de *TControl*, independentemente de se chamar o método *BringToFront*;

*ClientToParent*: translada as coordenadas do ponto informado para as relativas ao controle-pai. No exemplo da **Listagem 5** ele é usado para mover corretamente o botão pelo formulário, já que os valores dos parâmetros X e Y são relativos ao controle em si, mas as propriedades *Top* e *Left* são representadas no plano do formulário-pai;

*ClientToScreen*: Como no método anterior, translada o ponto passado para o sistema de coordenadas da tela;

*ParentToClient* e *ScreenToClient*: fazem o oposto dos dois últimos métodos apresentados, respectivamente;

*Invalidate*: força o redesenho total do controle. Se mais de uma região dentro do controle precisar ser redesenhada, forçará toda a janela a se desenhar novamente, evitando que o efeito de *flickering* ocorra;

*Update*: processa quaisquer mensagens de desenho do controle, forçando seu redesenho imediatamente. Deve ser usado quando queremos atualizar algum controle visual, durante um processamento demorado, por exemplo;

*Repaint*: chama *Invalidate*, seguido de *Update* para redesenhar completamente o controle de forma imediata;

*Refresh*: como implementado em *TControl*, simplesmente chama *Repaint*;

*Perform*: serve para simular o envio de uma mensagem do Windows para o controle, geralmente para se obter uma funcionalidade não publicada.

## TWinControl

Essa classe engloba todos os controles que são *wrappers* (camadas entre o código nativo e as aplicações que precisam utilizá-lo) dos objetos do Windows, como *Edits*, *CheckBoxes*, *RadioButtons*, *Memos*, *Menus* etc.

Os controles descendentes de *TWinControl* podem processar as entradas do teclado e, logo, receber foco. Além disso, podem conter outros controles, fechando o conceito de container visual, iniciado por *TControl*. Assim, controles derivados daquela classe podem estar contidos em controles derivados de *TWinControl*, mas não vice-versa.

Diferentemente da classe apresentada no tópico anterior, onde as rotinas de desenho dos controles precisa ser implementada, os controles descendentes de *TWinControl* já fazem isso de forma automática, pois é o próprio Windows que cuida de desenhá-los. Por serem *wrappers* de objetos, a camada inferior do código quase nunca é vista, e essa é uma das razões que tornou a VCL tão popular.

Toda a comunicação com a API do Windows, tanto para renderizar quanto para processar as mensagens dos controles é feita de forma transparente pela VCL. Se for necessário "pular" o *wrapper*, *TWinControl* oferece a propriedade *Handle*, que é um atalho para o objeto nativo do sistema operacional.

Isso garante que funcionalidades não implementadas pela VCL possam ser realizadas, como acessar propriedades e métodos não publicados dos objetos do Windows. Dentre os atributos de *TWinControl*, podemos destacar:

*Focused*: verifica se o controle tem ou não o foco da aplicação;

*GetTabOrderList*: retorna a lista dos controles filhos desse, ordenados por seus índices de tabulações;

*SelectFirst*: foca o primeiro controle filho definido pela ordem de tabulação;

*SelectNext*: foca o próximo controle filho. É possível dizer se a busca será para frente ou para trás, e se será respeitado o valor da propriedade *TabStop*. Como exemplo, inicie uma nova aplicação e coloque diversos *Edits* no formulário. Implemente então o evento *OnKeyPress* de algum deles para o código da **Listagem 6**.

**Listagem 6. Movendo o foco com Enter**

```
procedure TForm1.Edit1KeyPress(Sender: TObject;
  var Key: Char);
begin
  if Key = Chr(VK_RETURN) then
  begin
    SelectNext(Sender as TWinControl, True, True);
    Key := #0;
  end;
end;
```

Agora atribua aos demais *Edits* o mesmo manipulador de evento. Rode a aplicação e veja que o foco das caixas de texto será movido de acordo com o pressionamento da tecla ENTER;

*SetBounds*: permite informar a posição e o tamanho do controle de uma única vez, isso é, *Left*, *Top*, *Width* e *Height*, respectivamente.

## TGraphicControl

Essa classe serve de base para controles conhecidos como *lightweight* (leves). Uma vez que eles não englobam objetos visuais do Windows, eles são mais rápidos e consomem menos recursos. *TGraphicControl* não aceita entradas do teclado e não pode conter outros controles.

Em contrapartida, ela fornece a propriedade *Canvas*, que dá acesso à "superfície" do controle, permitindo desenhar diretamente nela. Esse desenho é realizado pelo método *Paint*, que deve ser implementado nas classes descendentes da forma desejada. Os controles *Image* e *PaintBox* são alguns exemplos que herdam de *TGraphicControl*.

## TCustomControl

E se eu quisesse herdar de *TWinControl*, para me beneficiar de suas funcionalidades, e precisasse implementar meu próprio algoritmo de desenho? A resposta a essa pergunta é o título do tópico. *TCustomControl* fornece todas as características de *TWinControl*, e também a propriedade *Canvas*, que, como em *TGraphicControl*, provê acesso à área de desenho do controle. Além da propriedade, também é oferecido o método *Paint*, que tem a mesma função da classe citada no tópico anterior.

## Conclusão

Vimos neste artigo as principais classes que compõe a estrutura da VCL. Todos os demais componentes são baseados nelas, direta ou indiretamente, cada um implementando determinadas características, estendendo outras etc. Dominar o funcionamento dessas classes pode ajudar a resolver problemas triviais, como qual método chamar para realizar uma determinada tarefa ou evitar escrever uma propriedade que já existe em uma classe ascendente, por exemplo.

Como leitura complementar, eu sugiro uma longa olhada no Help do Delphi. Pesquise por cada classe apresentada aqui e estude seus métodos, propriedades e eventos. Se você tiver inclinação, abra o código-fonte da VCL, disponível com o Delphi, e tente entender como ele funciona.

Com certeza isso aumentará consideravelmente seu conhecimento sobre a linguagem. Um forte abraço a todos e até a próxima! ■


**CLUBEDELPHI PLUS!**

Acesse agora o mesmo o portal do assinante ClubeDelphi (www.clubedelphi.net/portal) e assista a uma vídeo-aula de Luciano Pimenta que mostra como pintar ícones em componentes ListBox, Statusbar, ComboBox, usando Canvas, no endereço *http://www.devmedia.com.br/articles/viewcomp.asp?comp=2083*.

Em uma vídeo aula de Guinther Pauli, no endereço *http://www.devmedia.com.br/articles/viewcomp.asp?comp=570*, veja como criar um controle visual personalizado, descendente de TGraphicControl.



**CLUBEDELPHI PLUS!**

Acesse agora o mesmo o portal do assinante ClubeDelphi (www.clubedelphi.net/portal) e assista a uma vídeo-aula de Guinther Pauli, que mostra como criar um componente que automaticamente atualiza um ClientDataSet a cada n segundos, usando um timer interno, publicado através do uso de SetSubComponent:
http://www.devmedia.com.br/articles/viewcomp.asp?comp=986

# Análise, reparação e otimização de bancos IB/FB

**EVERSON BORGES VOLACO**
(everson@rhealeza.com.br)
é desenvolvedor e instrutor certificado Borland, com experiência em aplicações cliente/servidor, usando Delphi, Interbase e Oracle. Possui três certificações oficiais Borland: Borland Delphi 7.0, Borland CaliberRM 6.0 e Borland StarTeam 6.0.

Neste artigo veremos como realizar análises, reparações e otimizações em banco de dados InterBase e Firebird. Utilizaremos no decorrer deste artigo duas ferramentas que permitem essas operações: *IB/FB Development Studio* (*www.sqlly.com*) e *Database Workbench* (*www.upscene.com*).

> **Nota:** As duas ferramentas utilizadas são pagas, porém você pode baixar uma versão trial no site do fabricante. Neste artigo utilizei a versão 5.0.5 do *Development Studio* e a versão 2.8.7 Pro do *Database Workbench*.

Devido ao fato das ferramentas possuírem um processo de instalação rápido e fácil não abordaremos isso no artigo. Iniciaremos diretamente a partir das interfaces das ferramentas começando a abordar as funcionalidades presentes no *IB/FB Development Studio*.

## IB/FB Development Studio

O *Development Studio* é agrupado em três seções: *Create*, *Optimize* e *Manage*. Iniciaremos abordando a seção *Create*. Através da opção *Database Designer* podemos criar projetos para criação do banco *IB/FB* através de scripts DDL (Linguagem de definição de dados) ou ainda através de diagramas de ER.

Um dos principais pontos dessa ferramenta é o seu grande número de assistentes. Podemos criar grande parte dos objetos de forma 100% visual. Através da janela *Database Designer* podemos realizar a engenharia-reversa de bancos de dados existentes, utilizar recursos de *Refactoring*, gerar documentação completa no formato HTML, criar *Stored Procedures* para operações básicas de *Insert*, *Update*, *Delete* e *Select*, realizar controle de versão dos objetos e ainda

"compilar" (opção de menu *File>Validate Project*) nosso banco de dados em busca de possíveis erros.

Como o foco do artigo não é demonstrar todas as funcionalidades das duas ferramentas, deixarei a cargo do leitor, descobrir todos os recursos disponíveis. Voltando à tela inicial da ferramenta, ainda dentro da seção *Create*, temos acesso à opção *Database Editor* que permite realizar diversas operações de análise no banco.

Ao clicar nessa opção pela primeira vez, aparecerá a janela *Database Manager* que permite configurar um alias para acesso ao banco de dados. Uma vez criado o alias podemos acessar a janela *Database Editor* dando um duplo clique em cima do mesmo (**Figura 1**).

Através dessa janela temos acesso à diversas opções para análise do banco de dados. A partir do menu *Database*, por exemplo, podemos ativar e desativar índices para fazer o re-balanceamento dos mesmos, compilar *Stored Procedures* e *Triggers* e ainda utilizar a opção *Check Database for code errors* para varrer o banco em busca de erros e *warnings* **(Figura 2)**.

Nessa janela também há opções de *Refactoring* que permitem, por exemplo, alterar o nome de um objeto e atualizar em todas as suas dependências. Dentro do menu *Tools* podemos acionar o *SQL Monitor* através do botão *Activate*. Desse ponto em diante qualquer instrução SQL realizada dentro da janela *SQL Editor* (*Tools>Mini SQL Editor*) será monitorada pelo *SQL Monitor*, que mostrará todas as operações realizadas pelo banco de dados para executar tal instrução.

Retornando a tela inicial vamos acessar a opção *Query Analyzer* disponível dentro da seção *Optimize*. Através dessa janela, podemos analisar e otimizar as instruções SQL, antes de adicionarmos as mesmas dentro de aplicações Delphi. O *Query Analyzer* permite a execução de instruções SQL e a visualização de gráficos e planos de execução referente a índices. Para mostrar as funcionalidades dessa opção vamos realizar um exemplo simples utilizando o banco de dados *Employee* que acompanha a instalação do InterBase e do Firebird.

Você pode registrar o banco *Employee* através da opção *Database Manager* disponível dentro do menu *View* da janela *Query Analyzer*. Após registrar o banco de dados, selecione a opção *File>New Query* para criar uma janela (*Query Editor*) para entrada de instruções SQL. Digite a seguinte instrução SQL:

```
select * from customer
```

Pressionando a tecla F9 ou através da opção de menu *Run>Execute* podemos executar a instrução SQL. O resultado da instrução aparecerá no formato de grid dentro da janela *Query Results (Ctrl + 2)*. Para visualizar estatísticas referente a instrução acesse a janela *Statistics (Ctrl+3)*. Dentro dessa janela podemos visualizar através de gráficos de barras informações sobre a utilização de índices, memória e páginas de dados.

Para nossa instrução de exemplo, em minha máquina, o comando demorou *31 ms* para ser executado e não utilizou nenhum índice para realizar a busca (barra do gráfico *Select statistic* foi apresentado em vermelho indicando que nenhum índice pode ser utilizado). Através da opção *Execution Plan (Ctrl+4)* podemos visualizar o plano que foi utilizado para a execução da instrução.

No caso do exemplo o plano foi CUSTOMER NATURAL. Volte a janela *Query Editor* e altere a instrução para:

```
select * from customer order by cust_no
```

Execute a instrução novamente e verá que dessa vez, como realizamos uma ordenação a partir do campo chave (CUST_NO) da tabela, o servidor de banco de dados utilizou o índice criado junto com a *Primary Key*. Acesse a janela *Statistics* e veja que agora o gráfico (*Select statistic*) mostra a barra na cor azul indicando uso de índice, porém o tempo para execução continua o mesmo.

Dentro da janela *Plan*, agora é mostrado o plano de execução utilizando o índice vinculado ao campo chave da tabela. Para finalizar esse primeiro exemplo altere a instrução SQL para:

```
select customer from customer order by cust_no
```

Executando novamente a instrução, verificamos que mais uma vez o índice foi utilizado e que o tempo para execução da instrução caiu quase pela metade (dê um *comitt* na instrução, através do menu *Transaction>Commit Transaction*, para que o tempo de execução seja alterado). A dica aqui é sempre que possível evitar o uso de instruções do tipo *select * from tabela*, claro. Dentro da seção *select* informe apenas os campos que trabalhará, evitando a utilização do caractere asterisco (*) para trazer todas as colunas da(s) tabela(s) que compõem a instrução.

Essa ferramenta pode ser bastante útil para análise e otimização de instruções SQL como as utilizadas, por exem-

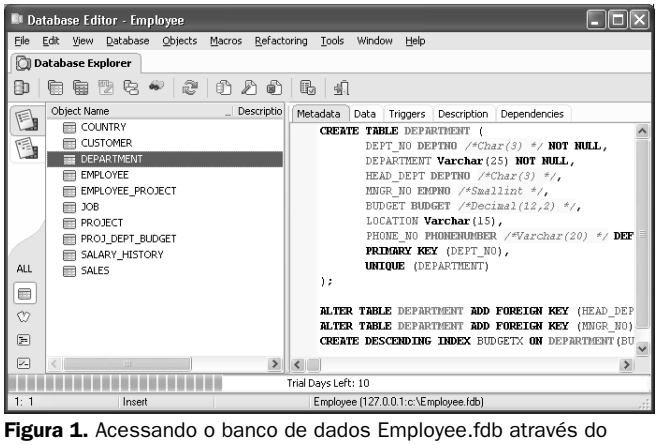



**Figura 1.** Acessando o banco de dados Employee.fdb através do Database Editor

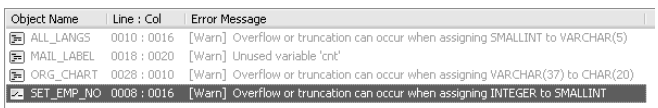



**Figura 2.** Verificando erros e warnings do banco de dados

plo, em consultas e relatórios que utilizam vários *join's* e filtros. A idéia é utilizar esse tipo de ferramenta como recurso para *tuning* de banco de dados.

**Nota:** Tuning significa a realização de ajustes em um SGBD para que alguma funcionalidade passe a funcionar melhor. Através desses ajustes buscamos um aumento de performance do banco de dados.

Abaixo segue algumas dicas relacionadas a análise para aumento de performance de banco de dados InterBase e Firebird.

**Nota:** Apesar do artigo ser focado nos banco de dados IB/FB, as dicas a seguir podem ser aplicadas a qualquer banco de dados relacional do mercado.

Pontos a serem gerenciados/analisados:

- Modelo de dados;
- Índices;
- Comandos SQL;
- Estrutura de hardware (CPU, I/O, memória etc.);
- Rede.

Para otimização de recursos de banco de dados devemos levar em conta alguns pontos:

- Criar índices somente quando for realmente necessário;
- Analisar a necessidade de criar índices para chaves estrangeiras;
- Reavaliar os índices existentes no banco de dados, adicionando ou removendo colunas do índice ou, se necessário removendo o mesmo;
- Recriar os índices que estejam com muita fragmentação ou com o nível da árvore *B*Tree* elevada;
- Identificar comandos SQL que possuam alto índice de consumo de recursos;
- Analisar utilização de ordenações (*order by*) e utilização de *join's*;
- Utilizar desde a fase de desenvolvimento do software, o banco de dados com grande volume de dados;
- Criar testes de repetição em cima das instruções SQL;
- Evitar a busca de registros que não serão utilizados na aplicação, isso é, linhas que são trazidas a aplicação e não são utilizadas pelo usuário;
- Realizar operações de *backup/restore* periodicamente;
- Realize testes de performance em máquinas cuja configuração de hardware seja equivalente ao do cliente do sistema.

Retomando a apresentação da ferramenta *Development Studio*, retorne para a tela inicial para acessar a opção *Database Comparer*, pra realizar a comparação e análise de dois diferentes bancos de dados. Essa opção pode ser bastante útil para você comparar diferentes versões do mesmo banco de dados, como, por exemplo, dentro de uma estrutura de matriz/filial.

Uma opção que sem dúvida pode nos ajudar no monitoramento e análise de nosso banco de dados é a opção *Performance Analyzer* disponível na seção *Optimize*. Através dessa opção temos acesso a janela *Interbase/Firebird Performance Analyzer* que permite visualizar gráficos de diversas informações do banco como memória, buffer, usuários conectados, páginas de dados, entre outros.

Para utilizar o *Performance Analyzer* selecione o banco de dados a ser monitorado através da aba *Databases* e clique sobre o botão *Monitor Selected Database*. Ao abrir uma nova aba contendo o alias do banco em questão clique no botão *Start* para iniciar a monitoração. Através dessa aba podemos ainda determinar quais gráficos queremos visualizar.

Uma vez iniciada a monitoração podemos utilizar a janela *Query Editor* ou ainda nossa própria aplicação Delphi para interagir com o banco de dados e acompanhar sua performance através dos gráficos. Veja na **Figura 2** o *Performance Analyzer* em execução.

A dica nesse ponto é utilizar as informações geradas pelos gráficos para tentar detectar as instruções SQL mais pesadas da nossa aplicação, afim de otimizá-las, aumentando assim o tempo de resposta de nossa aplicação para o usuário.

Como estamos abordando de maneira geral cada opção disponível no *Development Studio* vale comentar sobre as opções disponíveis na seção *Manage* da tela principal do produto. Através da opção *Administrative Console* podemos realizar operações de *backup/restore*, conceder ou revogar permissões de acesso e ainda extrair estatísticas do banco de dados e realizar validações para tentativas de recuperação de bancos corrompidos.

Ao entrar na janela *Interbase/Firebird Administrative Console*, basta selecionar o banco de dados que deseja verificar através da aba *Databases* e realizar as operações desejáveis através das opções disponíveis no menu principal. A partir da opção *Permissions>Grant Manager* podemos definir toda as permissões de acesso aos objetos de forma totalmente visual.

Selecionando a opção de menu *Database>Database Statistics* podemos gerar informações estatísticas gerais do banco sobre utilização de índices, páginas de dados, relacionamentos, entre outros (**Figura 3**).

Voltando a tela principal, temos acesso a última opção disponível, *Maintenance Service Manager* presente dentro da seção *Manage*. Nela, podemos agendar tarefas para realização de backup de banco de dados com opção de configuração de SMTP para recebimento de e-mail sobre os serviços ou ainda agendar a execução de scripts em geral.

## Database Workbench

A segunda ferramenta que abordaremos a fim de mostrar recursos para análise e otimização de banco de dados IB/FB será o *Database Workbench*, que apesar de não ser direcionada para uso exclusivo com IB e/ou FB como o *Development Studio*, possui diversas opções que podem auxiliar.

Após instalar a ferramenta você pode acessar a mesma

através do menu iniciar do Windows ou a partir do ícone de atalho criado no desktop. O primeiro passo para poder utilizar a ferramenta é registrar o servidor de banco de dados. Para isso, selecione o botão *Register Server* ( ) disponível na janela *Enterprise Manager* (lado esquerdo). Ao iniciar o assistente de registro de servidor (*Register Server Wizard*) selecione o banco de dados que utiliza no campo *Server type*.

**Nota:** Para a demonstração do *Database Workbench* utilizei o Firebird 1.5.2.

Clicando no botão *Next* passamos a janela *Server Registration Properties* (**Figura 4**) que permite configurar o servidor local ou remoto para acesso.

Ainda na janela *Enterprise Manager* selecione a opção *Databases* e a partir do menu de contexto execute a opção *Register Database* para acessar o assistente de registro de banco de dados. Clique no botão *Next* dentro da janela *Register Database* para iniciar a configuração para acesso.

Entre com as informações do banco de dados a ser acessado. Clique em *Next* e na seqüência no botão *Finish* para registrar o banco e adicioná-lo ao *Enterprise Manager*. Dê um duplo clique no *alias* criado para conectar e acessar os objetos do banco de dados recém registrado. O *Database Workbench* possui editores visuais para a criação e manutenção de todos os objetos presentes no IB/FB.

Outro ponto forte da ferramenta são as opções destinadas a importação e exportação de dados. Você pode ter acesso a essas opções através do menu *Tools*. Através da aba criada para acesso ao banco de dados registrado podemos acessar os recursos de manutenção e gerenciamento utilizando a opção *Management & Maintenance*. Dentro do nó *Maintenance* podemos extrair estatísticas do banco de dados (*Database statistics*), realizar limpeza no mesmo (*Perform Sweep*) e ainda operações de *backup* e *restore*.

Existem ainda algumas opções de análise que estão disponíveis através do menu de contexto da opção *Management & Maintenance*. Clicando com o botão direito sobre a opção temos acesso a recursos como recálculo de índices, compilação de *Stored Procedures*, migração de dados e comparação de banco de dados. Para controle de permissões de acesso a ferramenta traz a opção *Grant Manager* disponível no menu *Tools*, onde podemos, de forma visual, conceder ou revogar direitos de acesso aos usuários para todos os objetos do banco de dados.

O *Database Workbench* permite ainda a análise de execução de instruções SQL. Para criar e testar instruções SQL utilize a opção *SQL Editor* disponível no menu *Editors*. Dentro da janela *SQL Editor* podemos entrar com instruções simples (aba *Single SQL*) ou ainda com scripts (aba *SQL Script*). Para que possamos visualizar os recursos disponíveis no *SQL Editor* vamos criar um pequeno exemplo utilizando o banco *Employee*.

Registre o banco dentro da janela *Enterprise Manager* e acesso o *SQL Editor* (*F2*). Entre com a seguinte instrução SQL:

```
select c.customer, s.order_date, s.total_value
from customer c, sales s
where sales.cust_no = customer.cust_no
order by order_date desc
```

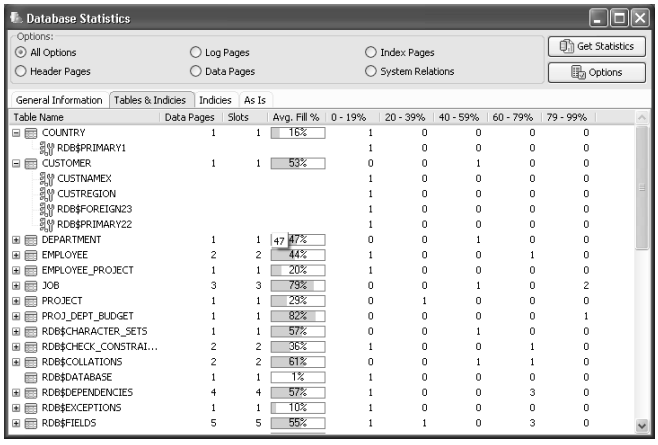



**Figura 3.** Visualizando informações estatísticas do banco de dados

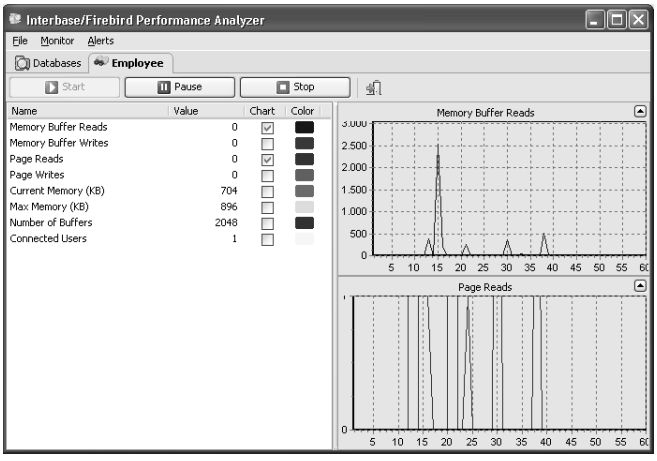



**Figura 2.** Monitorando o banco de dados Employee através do Performance Analyzer

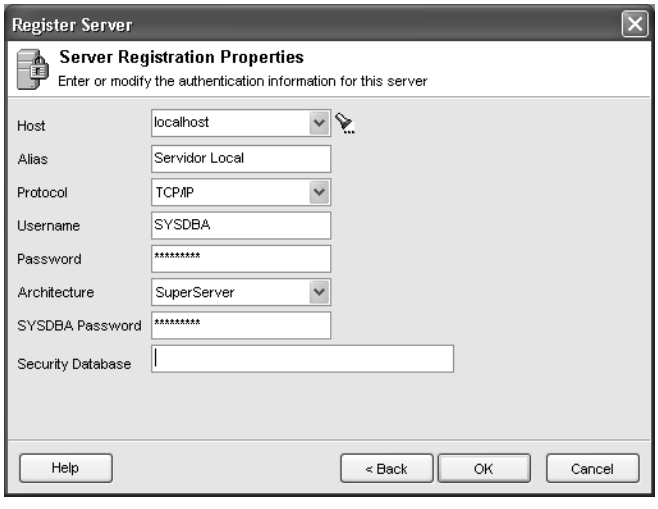



**Figura 4.** Registrando um servidor Firebird para acesso aos bancos de dados

Para executar a instrução SQL pressione a tecla F9. Os registros retornados pelo *select* serão exibidos dentro da aba *Results* na parte inferior da janela. Clicando sobre a aba *Plan Analysis* podemos visualizar o diagrama que representa o plano de execução utilizado para a instrução SQL (**Figura 5**).

Selecionando cada etapa do fluxo, temos detalhes da mesma, dentro da janela localizada na parte direita inferior. Clicando na aba *Statistics* temos acesso a informações sobre as tabelas utilizadas na instrução SQL, assim como o consumo de memória, uso do *buffer* e índices utilizados. No exemplo, visualizamos que para a tabela CUSTOMER não foi utilizado índice (*NATURAL*) enquanto que para a tabela *Sales* o índice relacionado a *Foreign Key* foi utilizado na pesquisa.

Caso você não tenha muita experiência na construção de instruções SQL você pode utilizar a opção *Visual Query Builder* disponível na barra de menu da janela *SQL Editor*. Através da janela *Visual Query Builder* podemos criar de forma visual a instrução SQL, configurando facilmente filtros (*where*), agrupamentos e ordenação (**Figura 6**).

Independente da ferramenta utilizada, você possui hoje diversas maneiras de detectar e corrigir gargalos em suas aplicações que utilizam banco de dados relacional. Nas duas ferramentas apresentadas neste artigo, vimos que no caso do InterBase e Firebird temos acesso a diversos recursos, muitos deles, presentes em grandes bancos de dados como Oracle e DB2.

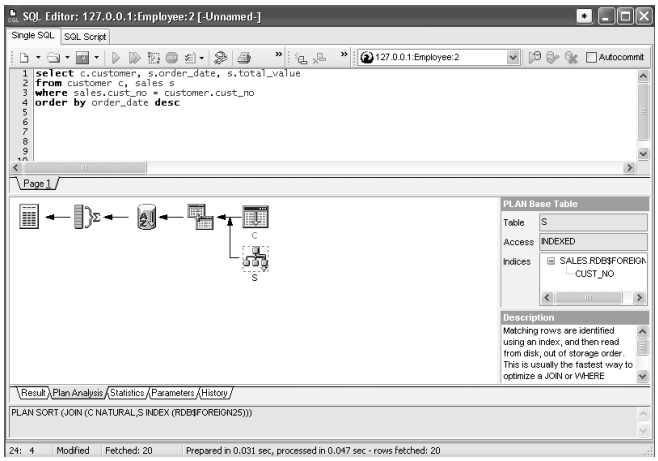



**Figura 5.** Fluxo e o plano de execução relacionado aos índices presentes na instrução SQL

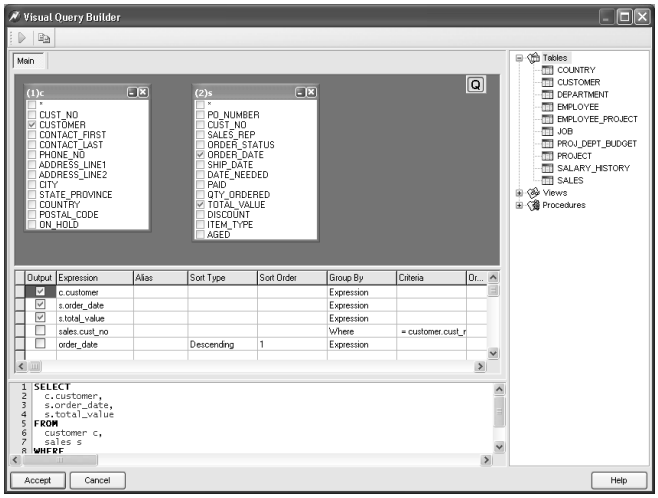



**Figura 6.** Criando e modificando instruções SQL através do Visual Query Builder

> **Saiba mais!** Você pode ter acesso a mais informações sobre técnicas e otimizações de bancos IB/FB no artigo *Acesso ao InterBase e Firebird*, publicado na edição 61. Referente a ferramentas de gerenciamento, na edição 60 temos um artigo sobre o *IBExpert*. Para finalizar, há um artigo bem abrangente sobre performance de banco de dados InterBase, publicado na edição 72 (*InterBase Performance Monitor*).

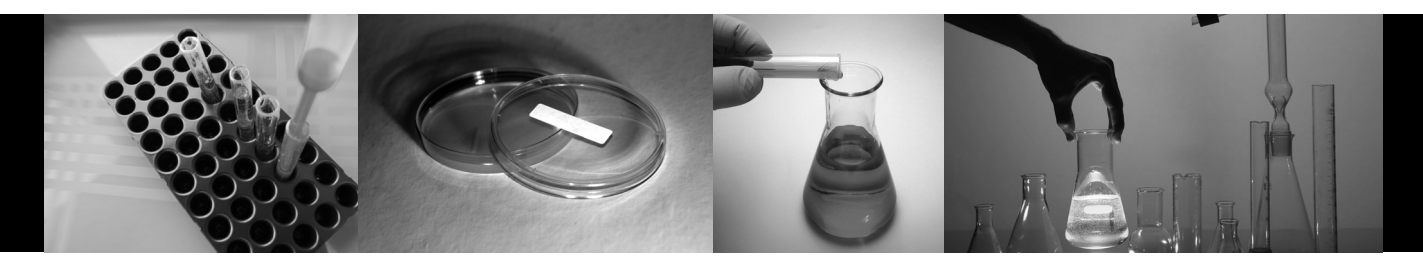
## Conclusão

Vimos neste artigo, duas ferramentas que podem ser utilizadas para gerenciar e realizar manutenção de banco de dados InterBase e Firebird. Além dessas, existem hoje no mercado uma infinidade de ferramentas, pagas e gratuitas, que podem auxiliar nas questões de análise, reparação e otimização de banco de dados. Um grande abraço e até a próxima. ■

### CLUBEDELPHI PLUS!

Acesse agora o mesmo o portal do assinante ClubeDelphi (www.clubedelphi.net/portal) e assista a uma vídeo-aula de Vídeo aula de Everson Volaco, que mostra como criar o banco de dados do sistema SysCash.

*http://www.devmedia.com.br/articles/viewcomp.asp?comp=2608*



# Complete sua coleção!

Feita para Desenvolvedores

SQL magazine

Escolha 03 edições e pague **R$ 6,90** em cada revista.

**20**

- Mapeamento Objeto Relacional. Aprenda a construir um mapeador O/R em .NET.

- Warehouse Builder. Construa uma data warehouse com a ferramenta ETL da Oracle-Partel

- UML. Aprenda passo a passo como construir um diagrama de sequências.

- Otimização. Influenciando o otimizador de consultas.

**21**

- Normalização. Aprenda a organizar melhor o esquema de seu banco de dados

- PostreSQL. Saiba o que são rules e como utilizá-las.

- SQL Server 2000. Entenda os modelos de recovery.

- InterBase 7.5. Quando e porque migrar.

- Particionamento do Oracle. Perigos e armadilhas Parte II.

**22**

- Backup. Saiba como proteger seus dados no Oracle e no SQL Server.

- UML. Conheça algumas opções de ferramentas livres - Parte II.

- Homologação de dados. Saiba como avaliar a qualidade de seu modelo de dados.

- Especial. Acessando servidores Oracle a partir do SQL Server 2000.

**23**

- Proteja-se. Saiba como funciona o SQL Injection e aprenda a lidar com ele no SQL Server.

- Administrando o Oracle. Conheça o Oracle Enterprise Manager.

- Backup. Saiba como efetuar o restore de Bds no SQL Server.

- Modelagem. Utilizando padrões na elaboração do projeto de banco de dados.

**24**

- Java + BD + Web. Trabalh forma integrada com: MySQ Tomcat e Java com NetBea

- Oracle 10g. Aumente a disponibilidade de seu SGB com o automatic Storage Management.

- PostgreSQL. Veja na práti como efetuar replicação en servidores PostgreSQL.

**25**

- Clustering no SQL Server. Saiba para que serve e como funciona essa tecnologia de redundância.

- IBExpert. Configure e administre base de dados no Firebird e interbase.

- Caché. trabalhe com objetos diretamente no banco de dados.

- Oracle Discoverer. A prenda passo a passo a trabalhar com a ferramenta de análise de dados da Oracle.

**26**

- Mineração de dados. Entenda como ocorre a descoberta de informações em base de dados.

- IBExpert. Administrando o InterBase/Firebird. Conheça algumas características avançadas.

- UML. Ferramenta de modelagem free com engenharia reversa.

- MySQL. Transforme-se em um especialista na otimização de consultas - Parte II.

**27**

- MySQL. Veja como rodar um banco de dados MySQL em várias máquinas ao mesmo tempo!

- ER/Studio. Conheça esta excelente ferramenta para modelagem de dados.

- SQL. Descobrindo associações entre pares de linhas.

- SQL Server. Veja como enviar e-mails anexos no SQL Server sem o uso do SQL Mail.

**28**

- Dúvidas frequentes em bancos de dados.

- Desafio SQL. Construindo um modelo para um jogo de futebol on-line.

- Index Tuning Wizard. Como produzir tunings automáticos.

- Erwin conheça esta ferramenta de modelagem E/R.

**29**

- Oracle FLASHBACK. Recupere informações para u determinado ponto do tempo

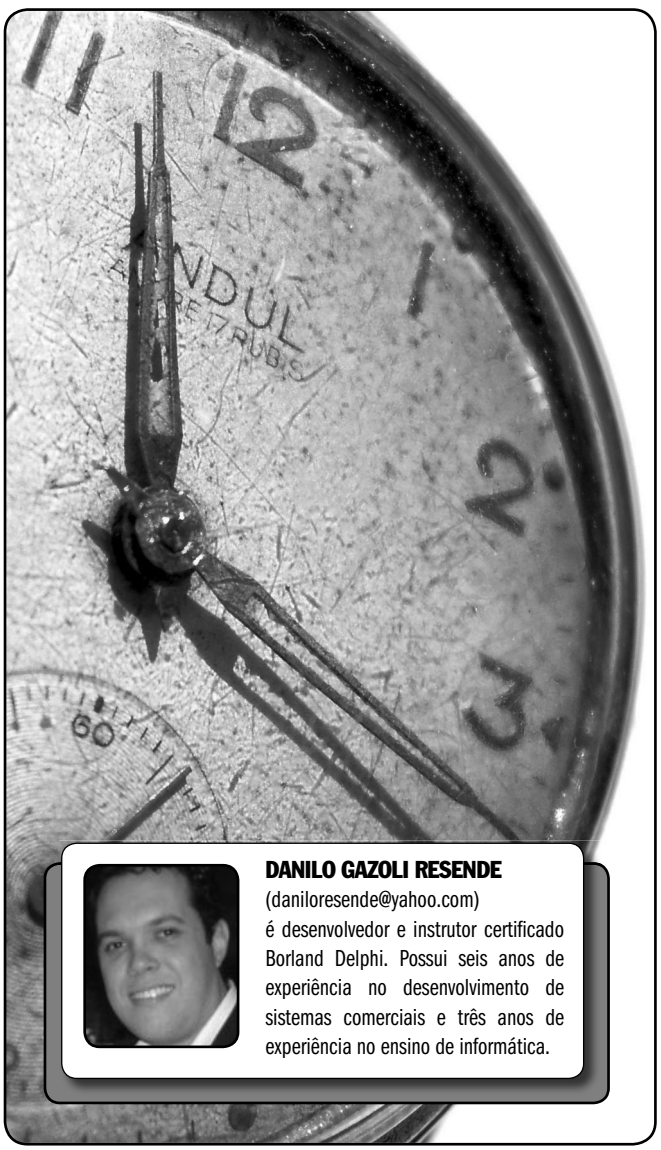
- SQL Server. Descubra se vo esta trabalhando corretamen com o SQL Server.

- InterBase / Firebird. Trabalhando com stored procedures e triggers.

**www.devmedia.com.br/msdn/anteriores**

ou pela nossa central de atendimento: (21)2220-5375

# Criando formulários dinamicamente a partir de registros em um BD





**DANILO GAZOLI RESENDE**
(daniloresende@yahoo.com)
é desenvolvedor e instrutor certificado Borland Delphi. Possui seis anos de experiência no desenvolvimento de sistemas comerciais e três anos de experiência no ensino de informática.

Um dos recursos interessantes na linguagem Pascal é a possibilidade de criar controles mesmo quando não sabemos de qual classe iremos instanciar. Através desse recurso, podemos entender a forma pela qual o Delphi trabalha em seu ambiente integrado de desenvolvimento quando clicamos em um *SpeedButton* da paleta de componentes e em seguida no formulário onde desejamos adicionar o mesmo.

A partir dos exemplos mostrados neste artigo, desenvolveremos um componente que fará a conexão a um banco de dados que contém os registros dos controles a serem criados no novo formulário, bem como uma aplicação exemplo desse novo componente.

## Preparação do banco de dados

Antes de começarmos a criar o componente, precisamos fazer as configurações necessárias no banco de dados, como criar as tabelas do "Dicionário de Formulários" e as tabelas que farão parte do sistema propriamente dito, que utilizaremos como teste para criar os controles desejados.

O banco de dados escolhido foi o Firebird, por ser gratuito e de fácil conexão através da tecnologia dbExpress. Chamaremos o conjunto de tabelas responsáveis pelos registros dos formulários dinâmicos como Dicionário de Formulários, representado por duas tabelas: "Formulario" e "Controle".

Na tabela *Formulario* incluiremos os campos principais para a criação do formulário e na *Controle*, como o nome sugere, incluiremos os campos para a criação dos controles do formulário. Na **Listagem 1** temos o código para criação das tabelas.

Nesse ponto, precisamos entender a necessidade dos campos da tabela *Controle*:

• *Ordem*: será a ordem na qual o controle será criado;

• *DisplayLabel*: para as classes *TField* será criado um *Label* adicional para mostrar o título do campo, para os demais controles visuais será utilizado para configurar o *Caption*;

• *Field*: representa o campo da tabela;

• *Control*: representa o controle a ser criado para aquele campo;

• *FieldMask*: a máscara atribuída ao campo;

• *FieldType*: pode ser *ftInteger* para campos do tipo inteiro ou *ftString* para campos do tipo *string*;

• *FieldKind*: o tipo do campo, no nosso exemplo utilizaremos *fkData* e *fkLookup*;

• *LookupTable*: para o *FieldKind* do tipo *fkLookup*, se faz necessário configurar qual tabela será utilizada para a pesquisa, bem como *KeyFields*, *LookupKeyFields* e *LookupResultField*;

• *ControlWidth* e *ControlHeight*: servem para especificar a largura e a altura respectivamente.

O próximo passo é incluir os registros nas tabelas criadas anteriormente, conforme a **Listagem 2**.

**Listagem 1. Código de criação das tabelas**

```
CREATE TABLE FORMULARIO (
  IDFORMULARIO INTEGER NOT NULL PRIMARY KEY,
  NOME VARCHAR(30) NOT NULL,
  FORMWIDTH INTEGER,
  FORMHEIGHT INTEGER,
  CAPTION VARCHAR(60)
);

CREATE TABLE CONTROLE (
  IDCONTROLE INTEGER NOT NULL,
  IDFORMULARIO INTEGER NOT NULL
    REFERENCES FORMULARIO(IDFORMULARIO),
  ORDEM INTEGER,
  DISPLAYLABEL VARCHAR(50),
  FIELD VARCHAR(50),
  CONTROL VARCHAR(50),
  FIELDMASK VARCHAR(40),
  VISIBLE CHAR(1),
  FIELDTYPE VARCHAR(20),
  FIELDKIND VARCHAR(20),
  LOOKUPTABLE VARCHAR(50),
  KEYFIELDS VARCHAR(40),
  LOOKUPKEYFIELDS VARCHAR(40),
  LOOKUPRESULTFIELD VARCHAR(40),
  CONTROLWIDTH INTEGER,
  CONTROLHEIGHT INTEGER
)
```

**Listagem 2. Código para inserção dos dados**

```
INSERT INTO FORMULARIO VALUES (1,'CLIENTE', 450, 450,
  'Cadastro de Clientes');
INSERT INTO CONTROLE VALUES (1, 1, 1, NULL, NULL,
  'TDBNavigator', NULL, 'S', NULL, NULL, NULL, NULL,
  NULL, NULL, NULL, NULL);
INSERT INTO CONTROLE VALUES (2, 1, 2, 'Código',
  'IDCliente', 'TDBEdit', NULL, 'S', 'fkData', NULL,
  'ftInteger', NULL, NULL, NULL, 50, NULL);
INSERT INTO CONTROLE VALUES (3, 1, 3, 'Nome', 'Nome',
  'TDBEdit', NULL, 'S', 'fkData', NULL, 'ftString',
  NULL, NULL, NULL, 200, NULL);
INSERT INTO CONTROLE VALUES (4, 1, 4, 'Idade',
  'Idade', 'TDBEdit', NULL, 'S', 'fkData', NULL,
  'ftInteger', NULL, NULL, NULL, 50, NULL);
INSERT INTO CONTROLE VALUES (5, 1, 5, 'Tefefone',
  'Telefone', 'TDBEdit', '!\(99\)0000-0000;1;_', 'S',
  'fkData', NULL, 'ftString', NULL, NULL, NULL, 80,
  NULL);
INSERT INTO CONTROLE VALUES (6, 1, 6, 'Sexo', 'Sexo',
  'TDBLookupComboBox', NULL, 'S', 'fkLookup', 'Sexo',
  'ftString', 'Sexo', 'IDSexo', 'Descr', 100, NULL);
INSERT INTO CONTROLE VALUES (7, 1, 7, 'Cidade',
  'IDCidade', 'TDBLookupComboBox', NULL, 'S',
  'fkLookup', 'Cidade', 'ftInteger', 'IDCidade',
  'IDCidade', 'Nome', 200, NULL);
INSERT INTO CONTROLE VALUES (8, 1, 8, NULL, NULL,
  'TDBGrid', NULL, 'S', NULL, NULL, NULL, NULL, NULL,
  NULL, 420, 150);
```

E, por fim, criaremos as tabelas básicas do sistema da aplicação de exemplo que servirão para os testes do componente a ser desenvolvido: Cliente, Sexo, Cidade, conforme a **Listagem 3**.

**Listagem 3. Tabelas para os exemplos**

```
CREATE TABLE CIDADE (
  IDCIDADE INTEGER NOT NULL PRIMARY KEY,
  NOME VARCHAR(50) NOT NULL);

CREATE TABLE SEXO (
  IDSEXO CHAR(1) NOT NULL PRIMARY KEY,
  DESCR VARCHAR(20) NOT NULL);

CREATE TABLE CLIENTE (
  IDCLIENTE INTEGER NOT NULL PRIMARY KEY,
  NOME VARCHAR(50),
  IDADE SMALLINT,
  TELEFONE VARCHAR(30),
  SEXO CHAR(1) REFERENCES SEXO(IDSEXO),
  IDCIDADE INTEGER REFERENCES CIDADE(IDCIDADE));
```

## O componente TDynForm

Finalmente vamos ao que interessa! Chamaremos o componente de "DynForm" que será descendente direto de *TComponent*, portanto não visual. No Delphi 7 acesse o menu *Component>New Component*, escolha *TComponent* como classe base e clique em *Next*. A **Figura 1** ilustra as configurações a serem feitas nessa etapa.

> **Nota:** Usamos o dbExpress como tecnologia para acesso a dados. Mas, com algumas alterações, você pode trabalhar com outra tecnologia, como ADO, BDE, IBX.

O *TDynForm* conterá três propriedades publicadas:

• *DataSource*: utilizado para ligar ao *DataSet* no qual os campos serão criados;

• *TableName*: representa a tabela e os controles a serem buscados no Dicionário de Formulários;

• *SQLConnection*: utilizado para conexão ao banco, uma vez que precisamos alimentar os *DataSets* internos utilizados para o *Lookup*.

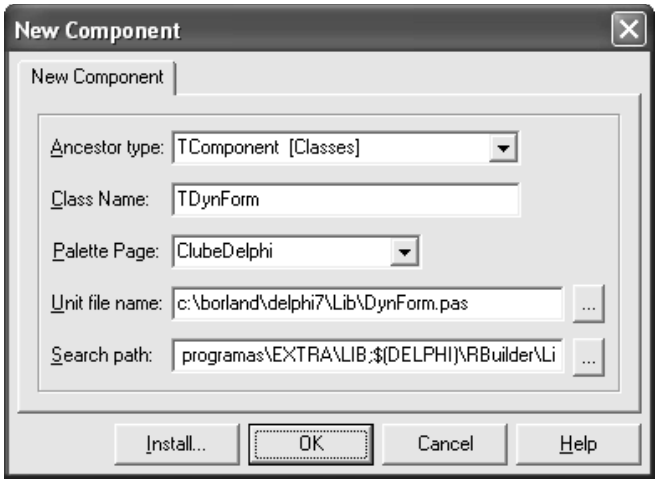



**Figura 1.** Configurações para criação do DynForm

Os principais campos e métodos serão:

• *FLookupDataSets*: trata-se de um vetor dinâmico de *TDataSet* que será utilizado para instanciar os *Lookups* utilizados nos controles do tipo *TComboBox*;

• *FDictDataset*: campo privado do tipo *TSQLDataSet*, utilizado para a busca dos registros do Dicionário de Formulários;

• *FForm*: campo do tipo *TForm*, que será instanciado e servirá de container para os controles;

• *LayoutControls*: método privado que cria os controles no container *FForm*;

• *Execute*: método público que será utilizado para criar o Formulário e os controles.

Na **Listagem 4**, temos o código parcial contendo a declaração da classe do componente.

**Listagem 4. Código parcial com a declaração da classe do componente**

```
uses SysUtils, Classes, SimpleDS, SqlExpr, DB, Forms,
  DBCtrls, DBGrids, Controls, TypInfo, StdCtrls,
  Buttons, DBClient;

type
  TDynForm = class(TComponent)
  private
    FLookupDataSets: Array of TSimpleDataSet;
    FIncLookup: integer;
    FDictDataset: TSQLDataSet;
    FTableName: string;
    FDataSource: TDataSource;
    FDataSet: TDataSet;
    FForm: TForm;
    FSQLConnection: TSQLConnection;
    procedure SetTableName(const Value: string);
    procedure LayoutControls;
    procedure OpenDictTable(SQL: String);
    procedure OpenDataSet;
    function FindLookupDataSet(LookupTable,
      LookupKeyFields: String): TDataSet;
    procedure ClickApplyUpdates(Sender: TObject);
    procedure SetSQLConnection(
      const Value: TSQLConnection);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
    procedure Execute;
  published
    property DataSource: TDataSource read FDataSource
      write FDataSource;
    property TableName: string read FTableName
      write SetTableName;
    property SQLConnection: TSQLConnection
      read FSQLConnection write SetSQLConnection;
  end;
```

Como visto, o "coração" do *TDynForm* reside no método *LayoutControls* (**Listagem 5**). Todo o código está bem comentado, para facilitar o entendimento do leitor.

**Listagem 5. Método LayoutControls do TDynForm**

```
procedure TDynForm.LayoutControls;
var
  { Representa a referência de classes }
  Reference: TControlClass;
  { Representa o controle que será criado }
  Instance: TControl;
  iTop: Integer;
  FField, FLookupField: TField;
  FLabel: TLabel;
  FSpeedButton: TSpeedButton;
begin
  if (FDictDataset.Active) and Assigned(FForm) then
  begin
    { Fetch inicial dos registros do Dicionário
      de Formulários e configuração de altura,
      largura e Caption do formulário }
    FForm.Height := FDictDataset.FieldbyName('FormHeight').AsInteger;
    FForm.Width := FDictDataset.FieldbyName('FormWidth').AsInteger;
    FForm.Caption := FDictDataset.FieldbyName('Caption').AsString;

    FDataSet.Close;
    FDataSet.Fields.Clear;
    iTop := 8;
    { FSQLDaSet é unidirecional, portanto deve-se
     fazer a leitura de uma vez. FDictDataSet traz a
     definição dos campos e controles do Dicionário
     de Formulários }
    FDictDataset.First;
    while not FDictDataset.Eof do
    begin
      { Criação dos objetos TField }
      FLookupField := nil;
      FField := nil;
      case TFieldType(GetEnumValue(TypeInfo(
        TFieldType), FDictDataset.FieldbyName(
        'FieldType').asString)) of
        ftString: FField := TStringField.Create(Self);
        ftInteger: FField :=
          TIntegerField.Create(Self);
      end;

      if Assigned(FField) then
      begin
        { Configurações do campo }
        FField.Name := 'Field' + FDataSet.Name +
          FDictDataset.FieldbyName('Field').AsString;
        FField.FieldName := FDictDataset.FieldbyName(
          'Field').AsString;
        FField.DisplayLabel := FDictDataset.FieldbyName(
          'DisplayLabel').AsString;

        if FDictDataset.FieldbyName(
          'FieldMask').AsString <> '' then
          FField.EditMask := FDictDataset.FieldbyName(
          'FieldMask').AsString;
        FField.Visible := FDictDataset.FieldbyName(
          'Visible').AsString = 'S';
        FField.DataSet := FDataSet;

        if TFieldKind(GetEnumValue(TypeInfo(
          TFieldKind), FDictDataset.FieldbyName(
          'FieldKind').AsString)) = fkLookup then
        begin
          { Não mostrar o campo, mas sim seu lookup }
          FField.Visible := False;

          { Se for do tipo lookup, deve-se criar um
            campo lookup Adicional e configurar o
            DataSet lookup }
          FLookupField := TStringField.Create(Self);
          FLookupField.Name := 'LookupField' +
               FDataSet.Name + FDictDataset.FieldbyName(
               'Field').AsString;
          FLookupField.FieldName := 'Lookup' +
            FDictDataset.FieldbyName('Field').AsString;
          FLookupField.DisplayLabel :=
            FDictDataset.FieldbyName(
            'DisplayLabel').AsString;
          FLookupField.FieldKind := fkLookup;
          FLookupField.DataSet := FDataSet;

          FLookupField.KeyFields :=
            FDictDataset.FieldbyName(
            'KeyFields').AsString;
          FLookupField.LookupKeyFields :=
            FDictDataset.FieldbyName(
            'LookupKeyFields').AsString;
          FLookupField.LookupResultField :=
            FDictDataset.FieldbyName(
            'LookupResultField').AsString;
    { FindLookupDataSet é método que retorna o Lookup,
           seja através da criação de uma nova tabela ou
           na busca de uma existente no array
           FLookupDataSets }
```

```
        FLookupField.LookupDataSet :=
      FindLookupDataSet(FDictDataset.FieldbyName(
           'LookupTable').AsString,
           FDictDataset.FieldbyName(
           'KeyFields').AsString);
      end;
    end;
    { Com os campos criados, é hora de criarmos
     os controles }
    Reference := TControlClass(FindClass(
      FDictDataset.FieldbyName('Control').AsString));
     if Assigned(Reference) and (UpperCase(
       FDictDataset.FieldbyName(
       'Visible').AsString) = 'S') then
     begin
       { Criar o controle }
       Instance := Reference.Create(Self);
       Instance.Name := Reference.ClassName +
         FDictDataset.FieldbyName(
         'IDControle').AsString;
       Instance.Tag := FDictDataset.FieldbyName(
         'IDControle').AsInteger;
       { Especificar o pai, então o controle
         é mostrado }
       Instance.Parent := FForm;
       if Assigned(FField) then
       begin
         { Criar os labels que acompanham os campos }
         FLabel := TLabel.Create(Self);
         FLabel.Caption := FField.DisplayLabel;
         FLabel.Parent := FForm;
         FLabel.Top := iTop;
         FLabel.Left := ciLeft;
         iTop := iTop + FLabel.Height;
       end;
       Instance.Top := iTop;
       Instance.Left := ciLeft;
       Instance.Width := FDictDataset.FieldbyName(
         'ControlWidth').AsInteger;

       { Configuração específica dos controles a
        serem instanciados }
       if Instance.InheritsFrom(TDBNavigator) then
       begin
         TDBNavigator(Instance).DataSource :=
           FDataSource;
         if FDataSource.DataSet is
           TCustomClientDataSet then
         begin
           FSpeedButton := TSpeedButton.Create(Self);
           FSpeedButton.Parent := FForm;
           FSpeedButton.Top := iTop;
           FSpeedButton.Left := Instance.Left +
             Instance.Width;
           FSpeedButton.Height := Instance.Height;
           { ClickApplyUpdates, como sugere, é o
            método que submeterá as alterações e
            inserções de registros para o banco }
           FSpeedButton.OnClick := ClickApplyUpdates;
         end;
       end
       else
       if Instance.InheritsFrom(TDBEdit) then
       begin
         TDBEdit(Instance).DataField :=
           FDictDataset.FieldbyName('Field').AsString;
         TDBEdit(Instance).DataSource :=  FDataSource;
       end
       else
       if Instance.InheritsFrom(
         TDBLookupComboBox) then
       begin
         if Assigned(FLookupField) then
           TDBLookupComboBox(Instance).DataField :=
           FLookupField.FieldName;
         TDBLookupComboBox(Instance).DataSource :=
           FDataSource;
       end
       else
       if Instance.InheritsFrom(TDBGrid) then
       begin
         TDBGrid(Instance).DataSource := FDataSource;
         TDBGrid(Instance).Left := ciLeft;
         if FDictDataset.FieldbyName(
           'ControlHeight').AsString <> '' then
           TDBGrid(Instance).Height :=
             FDictDataset.FieldbyName(
             'ControlHeight').AsInteger;
       end;
       iTop := iTop + Instance.Height + ciDist;
     end;
     FDictDataset.Next;
   end;
  end;
end;
```

A "mágica" que faz os controles serem instanciados dinamicamente reside em um conceito chamado Referência de Classes. Os tipos de referências de classes são usados normalmente para tratar várias classes diferentes como uma única classe. No código da listagem anterior, temos uma variável chamada *Reference* do tipo *TControlClass*. Na unit *Controls*, *TControlClass* está declarada da seguinte forma:

```
type
  TControlClass = class of TControl;
```

Como todos os controles são descendentes da classe *TControl*, é possível utilizar um tipo de referência de classe para criar o controle desejado. A classe *TControlClass* é declarada como um tipo de referência de classe para *TControl*, portanto, qualquer ocorrência da *TControlClass* pode ser atribuída a qualquer classe que descenda de *TControl*.

No código do método *LayoutControls*, a variável *Reference* será determinada através da função *FindClass*. Uma vez determinada (*Assigned(Reference)*), utilizamos a variável *Instance* para finalmente criar o controle e configurar as propriedades *Name*, *Tag* e *Parent*.

Como vimos, o *FindClass* procura e retorna uma classe pelo nome. Conforme o Help do Delphi, *FindClass* procura as classes registradas no sistema de *Streaming*. Classes de formulários e classes de componentes que são referenciadas na declaração do formulário (variáveis de instância) são automaticamente registradas.

As demais classes podem ser registradas através da função *RegisterClass*, por isso, é necessário registrarmos as classes que instanciaremos no construtor e eliminarmos o registro no destrutor da classe (**Listagem 6**).

**Listagem 6. Métodos Create e Destructor da classe**

```
constructor TDynForm.Create(AOwner: TComponent);
begin
  inherited;
  { Campo interno que fará o fetch dos registros do
    Dicionário de Formulários }
  FDictDataset := TSQLDataSet.Create(Self);
  FDictDataset.CommandType := ctQuery;
  { Registrar as classes para que sejam utilizadas
    pelo FindClass }
  RegisterClass(TDBLookupCombobox);
  RegisterClass(TDBEdit);
  RegisterClass(TDBGrid);
  RegisterClass(TDBNavigator);
  FIncLookup:= 0;
end;

destructor TDynForm.Destroy;
var
  i: integer;
begin
  { Varre e limpa os ClientDataSets do Array }
  for i := 0 to FIncLookup - 1 do
  begin
    FLookupDataSets[i].Free;
    FLookupDataSets[i] := nil;
  end;
  FDictDataset.Free;

  { Desregistrar as classes }
  UnRegisterClass(TDBLookupCombobox);
  UnRegisterClass(TDBEdit);
  UnRegisterClass(TDBGrid);
  UnRegisterClass(TDBNavigator);
  inherited;
end;
```

A **Listagem 7** compreende os demais métodos da classe do componente.

**Listagem 7. Métodos auxiliares da classe TDynForm**

```
{ FDataSet representa a tabela que conterá os campos
  do Formulário a ser criado }
{ Declare no uses Dialogs }
procedure TDynForm.OpenDataSet;
begin
  try
    FDataSet.Open;
  except
    on E: Exception do
    begin
      ShowMessage('Houve um erro ao tentar abrir a '+
        'tabela: ' + #13#10 + E.Message);
    end;
  end;
end;

{ Método para abertura da tabela no dicionário }
procedure TDynForm.OpenDictTable(SQL: string);
begin
  if (FTableName <> '') and (Assigned(
    FDictDataset.SQLConnection)) then
  begin
    FDictDataset.CommandText := SQL;
    try
      FDictDataset.Close;
      FDictDataset.Open;
    except
        on E: Exception do
        begin
          ShowMessage('Houve um erro ao tentar abrir '+
            'a tabela do Dicionario: ' +#13#10+
            E.Message);
        end;
    end;
  end;
end;

{ Método para notificar caso algum controle
  tenha sido removido do Formulário }
procedure TDynForm.Notification(
  AComponent: TComponent; Operation: TOperation);
begin
  inherited;
  if (AComponent = FDataSource) and
    (Operation = opRemove) then
    FDataSource := nil;
  if (AComponent = FSQLConnection) and
    (Operation = opRemove) then
    FSQLConnection := nil;
end;

procedure TDynForm.SetTableName(const Value: string);
begin
  FTableName := Value;
end;

{ Chamado para executar todo o processo interno de
  criação do formulário e seus controles }
procedure TDynForm.Execute;
begin
  FDictDataset.Close;
  FForm := TForm.Create(Application);
  try
    if not (csDesigning in ComponentState) then
    begin
      FDataSet := FDataSource.DataSet;
      if (FDataSet <> nil) then
      begin
        OpenDictTable(Format(csSQLDictControls,
          [FTableName]));
        LayoutControls;
        OpenDataSet;
      end;
      FForm.ShowModal;
    end;
  finally
    FForm.Free;
    FForm := nil;
  end;
end;

{ Retorna o DataSet utilizado nos campos de
  lookup. A idéia consiste em: fazer a busca no array
  FLookupDataSets a fim de verificar se o mesmo já
  existe, caso não seja encontrado, um DataSet
  deverá ser criado }
function TDynForm.FindLookupDataSet(LookupTable,
  LookupKeyFields: string): TDataSet;
var
  iLkp: integer;
begin
  Result := nil;
  { Antes de criar um lookup, procurar se existe }
  for iLkp := 0 to Length(FLookupDataSets) -1 do
  begin
    if FLookupDataSets[iLkp].Name =
      LookupTable + LookupKeyFields then
    begin
      Result := FLookupDataSets[iLkp];
      break;
    end;
  end;

  if not Assigned(Result) then
  begin
    { Não achou, então deverá ser criado }
    SetLength(FLookupDataSets, FIncLookup + 1);
    FLookupDataSets[FIncLookup] :=
      TSimpleDataSet.Create(Self);
    FLookupDataSets[FIncLookup].Connection :=
      FSQLConnection;
    FLookupDataSets[FIncLookup].DataSet.CommandType :=
      ctQuery;
    FLookupDataSets[FIncLookup].DataSet.CommandText :=
      Format(csSelectLookup,[LookupTable]);
    FLookupDataSets[FIncLookup].Name := LookupTable +
      LookupKeyFields;
    FLookupDataSets[FIncLookup].Open;
    Result := FLookupDataSets[FIncLookup];
    FIncLookup := FIncLookup + 1;
  end;
end;

{ Método será direcionado para o evento OnClick do
  botão SpeedButton, para submeter os registros }
procedure TDynForm.ClickApplyUpdates(Sender: TObject);
begin
  if TClientDataset(FDataSet).ChangeCount > 0 then
    TClientDataset(FDataSet).ApplyUpdates(0);
end;

procedure TDynForm.SetSQLConnection(
  const Value: TSQLConnection);
begin
  FSQLConnection := Value;
  { Configura a conexão do DataSet interno }
  FDictDataset.SQLConnection := FSQLConnection;
end;
end.
```

Lembre-se também de adicionar a unit *TypInfo* à cláusula *uses* da unit e também as constantes da **Listagem 8**.

**Listagem 8. Constantes utilizadas para a classe TDynForm**

```
const
  ciLeft = 8;
  ciDist = 3;
  csSelectLookup = 'SELECT * FROM %s';
  { Essa constante representa o SQL a ser executado
    para buscar os registros do dicionário de
    Formulários }
  csSQLDictControls =
    ' SELECT '  +
    '   C.*, ' +
    '   F.FormWidth, '+
    '   F.FormHeight, ' +
    '   F.Caption ' +
    ' FROM Controle C ' +
    '   JOIN Formulario F ON C.IDFormulario = ' +
    '   F.IDFormulario '+
    ' WHERE UPPER(F.Nome) = UPPER(''%s'')' +
    ' ORDER BY Ordem';
```

## A aplicação exemplo

Uma vez codificado, crie um novo *package*, insira a unit e instale normalmente. Em uma nova aplicação, insira um *SQLConnection* e faça as configurações necessárias para acessar o banco que estamos trabalhando.

Inclua um *SimpleDataSet* ligado ao *SQLConnection*, um *DataSource* ligado ao *SimpleDataSet* pela propriedade *DataSet*, um *MainMenu* e por fim um *DynForm* que criamos. O formulário deve estar parecido com o da **Figura 2**.

No *DynForm*, configure o *DataSource* para *DataSource1*,

*SQLConnection* para *SQLConnection1* e *TableName* para "CLIENTE". No evento *OnClick* do item de menu referente ao cadastro de clientes será adicionado o seguinte código:

```
procedure TForm1.Clique(Sender: TObject);
begin
  DynForm1.Execute;
end;
```

Pronto, agora basta rodar o aplicativo e teremos o formulário em ação parecido com o da **Figura 3**.

## Conclusão

A programação mostrada aqui dispensa a necessidade de recompilação do executável, uma vez que permite que o controle da aplicação seja feito no lado servidor. Com os registros no Dicionário de Formulário podemos criar formulários dinâmicos e, ao fazer as manutenções necessárias, refletir imediatamente para o usuário final. ■

**Figura 2.** Formulário principal e os componentes da aplicação de exemplo

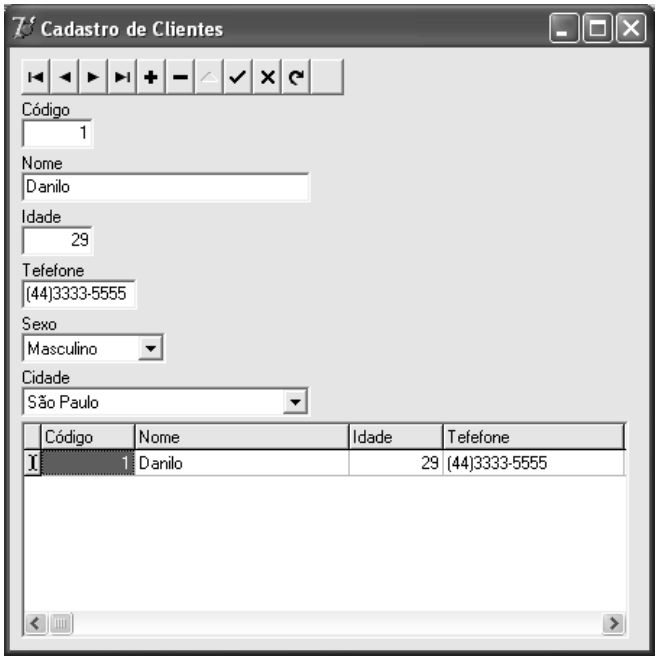



**Figura 3.** O DynForm em execução


Borland® E-Commerce

www.borlandshop.com.br

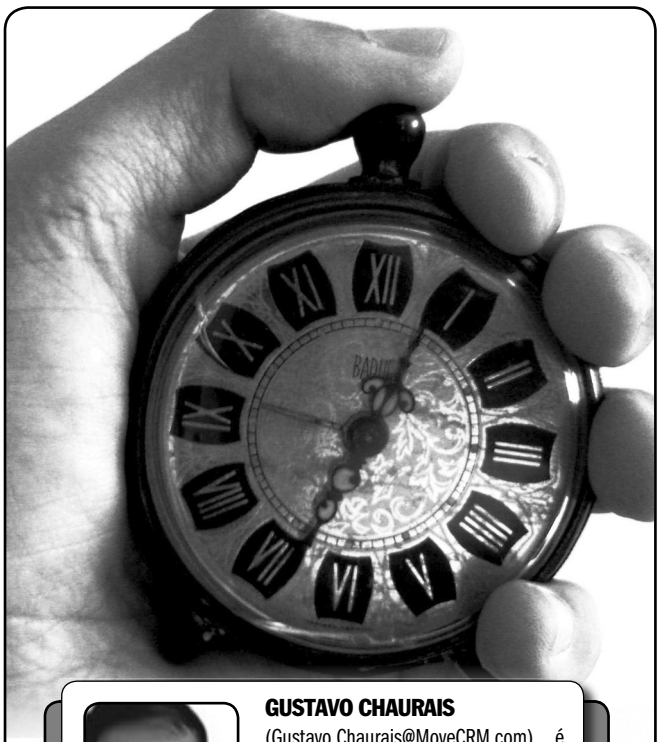
O seu canal direto de compras para produtos Borland

Mais Fácil

Mais Seguro

Mais Econômico

# Desenvolvimento Ágil e Seguro com Test-Driven Development



**GUSTAVO CHAURAIS**
(Gustavo.Chaurais@MoveCRM.com) é Borland Delphi 7 Advanced Product Certified, Borland Delphi 2005 for Win32 Product Certified, articulista/colunista e membro da comissão editorial da revista ClubeDelphi, analista/desenvolvedor da MoveCRM e graduando em Sistemas de Informação pela UFSC. Palestrante em grandes eventos como 4ª BorCon Brasil, ClubeDelphi TechWeek e DevMedia WebDays, é especializado em metodologias ágeis e no desenvolvimento de frameworks de aplicações orientados a objetos para projetos de grande e médio porte.

Quem nunca ouviu falar em metodologias ágeis? Um assunto tão em moda certamente não pode ser deixado de lado por nenhum desenvolvedor que queira estar atualizado. Esse "jeito novo" de fabricar software ajuda a compreender melhor os requisitos, estimar custos mais precisamente, garantir menos *bugs* etc.

Na situação em que vivemos, estamos aprendendo vagarosamente a não prender-se tanto à programação de regras de negócio, mas olhando também para um fator muito importante a esse tipo de produto: a sua qualidade final. Clientes não querem mais ver apenas o produto pronto e implantado, querem ver seu sistema funcionando e sem erros. Erros esses que podem levar sua empresa a uma situação de caos e fazê-los despender muito mais tempo e outros recursos em manutenção.

Mas como garantir que o sistema não terá erros? Veremos como o *Test-Driven Development* pode ajudar a diminuir a quantidade de *bugs* e ainda mais: aumentar a confiança no código pelos programadores, melhorar a qualidade da modelagem do projeto e estimar prazos e custos com muito mais precisão.

Como manipularemos código e nos basearemos fortemente em testes unitários, faremos uso do framework *DUnit*, uma ferramenta *free* e de código aberto, bastante utilizada em ambientes Delphi.

## Agile Development

Gosto sempre de fazer uma analogia ao explicar metodologias ágeis. Nos primórdios das teorias da administração, um dos fundamentos básicos era a formalização. Todo e qualquer tipo de processo era formalizado (escrito) para que, posteriormente, se pudesse aprender com experiências passadas e entender como funciona a empresa. Com isso nasceu a burocracia. Algo relativamente bom se for

seguido com disciplina, mas que pode atrapalhar e muito o andamento de uma organização.

Hoje, vemos que o foco é realmente outro. A valorização nas pessoas é muito grande quando comparada a antigamente. A motivação do colaborador é muito importante para que tudo ocorra bem. Quebras na estrutura hierárquica das organizações também são práticas que vêm ganhando espaço para diminuir as barreiras à comunicação entre os funcionários.

Se pararmos para analisar o desenvolvimento de software, algo muito mais novo veremos que a situação é muito semelhante. No início, não havia nenhuma forma de documentação. Após alguns anos, documentava-se tudo o que se podia especificar.

Hoje, estamos vendo na prática que a documentação não deve ser o fim, mas o meio para chegarmos ao produto final. Por isso, metodologias ágeis vêm ganhando espaço ao lado de metodologias pesadas, fortemente baseadas em documentação como, por exemplo, o *Unified Process*, definido pelo RUP da empresa *Rational*.

Outros motivos também levaram especialistas a pensar em processos mais ágeis na hora de desenvolver: *Incompreensão nos requisitos*, *Mudança freqüente nos requisitos*, *Falta de planejamento*, *Mudança de equipe* e *Bugs e regressões freqüentes*.

Para resolver esses e outros problemas, celebridades na área, reuniram-se e escreveram o *manifesto ágil*, que define diversos pontos para eliminar barreiras ao desenvolvimento de software. Talvez a metodologia ágil mais comentada e utilizada hoje seja o *eXtreme Programming*, que define alguns pontos muito importantes e interessantes. Outras metodologias ágeis conhecidas são: *Scrum*, *Crystal Clear*, *xUP*, *dX*, *MSF Agile* etc.

## Qualidade no software

Quando se fala em qualidade, logo temos em mente certificações ISO, CMMI etc. No entanto, esse tipo de controle de qualidade é aplicado basicamente aos processos utilizados, não ao produto final, que no caso é o sistema gerado.

O que podemos observar hoje, no mundo do software, é uma enorme movimentação de empresas de grande porte para que seu produto final não contenha erros de programação. Com isso nasce um novo cargo muito bem visto atualmente: o de *Quality Assurance* (QA), que traduzido também significa *Controle de Qualidade*.

Para se desenvolver um software é preciso levar em conta basicamente quatro quesitos: o tempo, os recursos (pessoal, financeiros etc.), os requisitos e a qualidade (**Figura 1**).

Se um desses pilares é afetado, certamente afetará os outros. Por exemplo, se o tempo está muito curto para a entrega do projeto, teremos que aumentar os recursos, contratando um funcionário, ou diminuir os requisitos e funcionalidades a serem implementadas, ou, o que quase sempre acontece, aumentar a velocidade na programação, deixando diversos *bugs* não corrigidos e causando enormes problemas.

Outra situação muito comum em um projeto é a seguinte: na hora da implantação do programa no cliente, muitos erros acontecem. O prazo quase sempre está estourado, ou quase lá, e o usuário continua pedindo novas funcionalidades, alterações de módulos existentes e correções. Isso leva a não testar o que é feito e podemos chegar facilmente a um ciclo vicioso (**Figura 2**).

Se temos mais pressão, certamente construiremos menos testes ou fazê-los menos abrangentes. Quanto menos testes escrevermos, maior a probabilidade de introduzirmos erros na aplicação. Quanto mais erros a aplicação possuir, maior vai ser a pressão por parte do cliente para que os *bugs* sejam corrigidos e o sistema passe a funcionar perfeitamente.

Ou seja, entramos em um ciclo muito comum e que se não for bem administrado pode levar um projeto a um estado de caos! Mas será possível não entrar em um ciclo evitando a tal situação de caos? A saída dada por *Kent Beck* em seu livro *"Test-Driven Development: By Example"* é muito simples: precisamos reverter a ordem. Quanto mais erros, mais testes devemos escrever. Quanto mais testes, menos *bugs* serão deixados no sistema. Com isso, o cliente fará menos pressão, o que nos deixará mais tranqüilos para codificar.

Existem diversos tipos de testes de software executados por *testers* e equipes de QA nas empresas. Os mais utilizados e importantes são: *Testes Unitários*, *Testes de Aceitação e Funcionais* e *Testes de Performance, Carga e Stress*.

**Figura 1.** Os quatro pilares no desenvolvimento de software

**Figura 2.** Ciclo vicioso de Kent Beck

## O Test-Driven Development

Agora que tivemos uma boa base sobre metodologias ágeis, qualidade e testes podemos começar a entender o que é e porque foi criada a metodologia *Test-Driven Development*. Elaborada basicamente por *Kent Beck* e formalizada no livro "*Test-Driven Development – By Example*", essa não é uma metodologia de testes, mas sim, de desenvolvimento de software que pode ser facilmente integrada a outras delas como o tão conhecido XP.

O TDD, além de dar a garantia de que não haverá *bugs*, ajuda, por exemplo, a saber quais serão as próximas 10 linhas de código a serem escritas! Os testes guiarão o desenvolvimento dizendo exatamente o que deve ser codificado.

Isso muda toda a maneira tradicional de escrever código! Você consegue se imaginar escrevendo um teste para algo que ainda não foi codificado? Por mais que isso pareça estranho, esse é um dos pilares dessa metodologia.

No caso do TDD, em vez de utilizarmos documentos escritos para modelagem e especificação da estrutura interna de um software, utilizaremos testes unitários. Um *unit test* seria para nós o que o diagrama de estados é para a UML. No caso do programador que procura exemplos, os testes mostraram aplicações práticas de cada elemento do modelo. E, se o teste for bem escrito, todas as funcionalidades estarão cobertas, apresentando 100% do que está sendo procurado.

O maior objetivo do *Test-Driven Development*, segundo *Ron Jeffries*, é o *Clean Code That Works* (código limpo e funcional). Se temos uma camada de testes garantindo que nada será quebrado, temos também total segurança para melhorar o *design* do código, algo que é feito com muita freqüência no TDD. Assim, além do programa estar sem erros para o cliente, também estará muito bom para nós, porque o código estará limpo e fácil de compreender.

A comunicação entre membros da equipe também será muito facilitada. Se temos testes sobre um código limpo, o entendimento sobre o mesmo será bastante grande, e isso acontecerá para todos os desenvolvedores, quase que independentemente de sua experiência.

Vale deixar claro que a documentação citada até o momento se refere à modelagem do código propriamente dita, e não à especificação do projeto como um todo. Por exemplo, se sua empresa utiliza *Use Cases*, poderá continuar usando-os, sem problema algum.

## Regras do Jogo

Nosso "mantra" será sempre o seguinte: *Red*, *Green*, *Refactor*. Lembre-se sempre disso. É nessa ordem que tudo acontecerá. Nosso ciclo vai ser o mostrado na **Figura 3**.

1. *Escrever um teste*: escrever apenas um teste simples, não necessariamente todos os testes do projeto;
2. *Fazer o mínimo para que o teste compile*;
3. *Ver o teste falhar*: essa etapa é muito importante, pois mostra que a funcionalidade não está implementada ou que o *bug*, retratado pelo teste, está realmente presente;
4. *Fazer o teste passar*: lembrando sempre de fazer o mínimo possível para que isso aconteça;
5. *Eliminar duplicações*: deve-se eliminar qualquer tipo de duplicação. Se o teste ainda estiver passando após uma modificação é porque a funcionalidade está funcionando, ou seja, podemos deixar o mais limpo possível sem problemas;
6. *Ver o teste passar*: para garantir que nada foi quebrado na fase de eliminação de duplicações;
7. *Inserir outro teste*: fechando o ciclo.

Faremos isso para cada pequena funcionalidade a ser implementada. Para cada teste novo seguiremos esse ciclo e, desse modo, o teste estará guiando a codificação.

## O DUnit

Para podermos automatizar os testes, precisaremos de ferramentas que auxiliem em tal tarefa. Para isso, existem diversas no mercado. Às vezes, até mais de uma por linguagem. Como vamos utilizar o Delphi 7 em nosso exemplo, vamos usar o *DUnit*. Se estivéssemos utilizando .NET, poderíamos utilizar o *NUnit*, que contém muitos outros recursos interessantes para potencializar ainda mais o desenvolvimento de testes unitários. A instalação do *DUnit* é muito rápida:

1. Faça o *download* do programa diretamente de sua página (endereço na seção *Links*);
2. Descompacte o arquivo para uma pasta de sua escolha;
3. Abra o diálogo *Environment Options* (menu *Tools>Environment Options*);
4. Vá para a aba *Library*;
5. No campo *Library Path*, clique no botão com as reticências;
6. Adicione a pasta *src*, que está dentro da pasta raiz do *DUnit* (do arquivo que você descompactou).

Após, o *DUnit* estará instalado e pronto pra ser utilizado.

**Figura 3.** Ciclo de desenvolvimento do TDD baseado no mantra Red, Green, Refactor

## Preparando os projetos para o exemplo

Depois de alguma teoria é hora de colocar a mão na massa e ver isso tudo funcionando. No Delphi, crie um novo projeto, remova a unit do formulário automaticamente gerado e salve-o como "ProjPilha.dpr".

Agora, clique em *View>Project Manager* para abrir o gerenciador de projetos, clique com o botão direito no único *Project Group* existente e selecione *Add New Project*. Adicione mais um projeto, remova seu formulário gerado e salve-o como "ProjPilhaTests.dpr". Salve também o *Project Group* como "ProjGroupPilha.bpg".

Agora que temos os dois projetos criados, vamos personalizar o segundo (*ProjPilhaTests.dpr*) para torná-lo nosso projeto de testes. Para isso, clique com o botão direito sobre o mesmo no *Project Group* e selecione *View Source*. Altere-o conforme a **Listagem 1**. Isso fará com que os testes registrados sejam executados em modo gráfico.

**Listagem 1. Alterando o projeto para um projeto de testes**

```
program ProjPilhaTests;

uses
  Forms,
  TestFramework,
  GUITestRunner;

{$R *.RES}

begin
  Application.Initialize;
  GUITestRunner.RunRegisteredTests;
end.
```

## Escolhendo os testes

Nosso exemplo será a construção *Test-Driven* de uma estrutura de dados do tipo "Pilha" para o armazenamento de números inteiros. Embora o Delphi já possua uma classe parecida nativamente, iremos recriá-la novamente para aprendermos as técnicas do TDD. Nossa classe terá basicamente quatro métodos:

- *Push*: adiciona um número na pilha;
- *Pop*: retira o número do topo;
- *Top*: retorna, sem retirar, o número do topo;
- *EstaVazia*: Retorna *True* se a pilha não possuir números.

Vamos iniciar o processo criando nosso primeiro *Test Case*, no caso, uma classe que conterá os métodos de teste. No projeto *ProjPilhaTests*, insira uma nova unit, salve-a como "PilhaTests.pas" e digite o código da **Listagem 2**.

**Listagem 2. Criando o primeiro teste**

```
unit PilhaTests;
interface
uses
  TestFramework;

type
  TPilhaTests = class(TTestCase)
  private
  public
  published
  end;

implementation

initialization
  RegisterTest(TPilhaTests.Suite);
end.
```

Em outras palavras, crie uma classe *TPilhaTests*, herdando de *TTestCase* e registre o teste na seção *Initialization*. Verifique se o que foi feito até agora está correto, compilando a aplicação (CRTL+F9). Nenhum erro deve ser disparado.

Antes de escrever nosso primeiro teste, um dos procedimentos básicos é escrever tudo o que pensamos em matéria de funcionalidades/regras de negócio, representados por testes, em uma *To-Do List*.

Pensando rapidamente podemos imaginar alguns testes:

- Adicionar um número à pilha e verificar se o mesmo é retornado no *Top*;
- Adicionar um número à pilha e verificar se o mesmo é retornado no *Pop*;
- Adicionar um número à pilha e verificar se a mesma não está vazia;
- Ao ser criada, a pilha deve estar vazia;
- Adicionar um número, retirá-lo com *Pop* e verificar se pilha está vazia novamente;
- Adicionar três números e verificar se, com *Pop*, retorna na ordem inversa;
- Adicionar três números e verificar se, com *Top*, só o primeiro número retornará;
- Tentar retirar com *Pop* quando a Pilha está vazia;
- Tentar retornar com *Top* quando a Pilha está vazia.

Não é necessário adivinhar todos os testes logo de início. Sempre que um novo vier à tona, escreveremos e esperaremos a melhor hora para introduzí-lo. Porém, imagino que com esses testes, cobriremos todas as funcionalidades de nossa *Pilha*.

## Escrevendo os testes

Vamos então escolher um primeiro teste a ser implementado. Devemos pensar em um teste simples e que nos forneça uma base para a criação de nossa classe. Examinando todos, vamos ficar com esse: "Ao ser criada, a pilha deve estar vazia".

Lembre-se sempre de nosso mantra: *Red*, *Green*, *Refactor*. Segundo nosso ciclo, vamos primeiro escrever o teste e fazê-lo compilar. Declare um procedimento de nome "TestPilhaVaziaCreate" na seção *published* da classe e o implemente conforme o código da **Listagem 3**.

**Listagem 3. Testando se a Pilha está vazia ao ser criada**

```
procedure TPilhaTests.TestPilhaVaziaCreate;
var
  UmaPilha: TPilha;
begin
  UmaPilha := TPilha.Create;
  CheckTrue(UmaPilha.EstaVazia);
  UmaPilha.Free;
end;
```

Nosso primeiro teste criará um objeto do tipo *TPilha* e fará uma verificação/asserção, garantindo que o retorno de *EstaVazia* é *True*.

Logicamente, se tentarmos compilar nosso projeto, seremos notificados de alguns erros. A classe *TPilha* não existe, nem tampouco a propriedade *EstaVazia*. Vá então ao *Pro-*

*jPilha* e adicione uma nova unit, salvando-a como "Pilha.pas". Modifique seu código conforme a **Listagem 4**.

**Listagem 4. Implementação da classe TPilha**

```
unit Pilha;

interface

type
  TPilha = class
  public
    EstaVazia: Boolean;
  end;

implementation
end.
```

Podemos observar uma variável com visibilidade pública em nossa classe, algo que, como sabemos, é expressamente proibido pelos puristas da Orientação a Objetos. Porém, vamos seguir com nossas regras e fazer somente o mais simples para compilar o projeto de testes.

Volte ao *ProjPilhaTests* e, através do menu *Project>Add To Project*, adicione a unit recém criada (*Pilha.pas*). Modifique também a unit *PilhaTests.pas*, adicionando *Pilha* à sua seção *uses*. Tente compilar a aplicação e tudo ocorrerá perfeitamente. Primeiro passo completo!

Agora rode a aplicação de testes e clique no botão com formato de *Play*. Você verá que o teste não passou como já era de se esperar (**Figura 4**).

O próximo passo então é fazer o mínimo de alterações possível para ver o teste passar. Na unit *Pilha*, declare um construtor público *Create* e implemente-o da seguinte maneira:

```
constructor TPilha.Create;
begin
  EstaVazia := True;
end;
```

Rode novamente o projeto de testes e pronto! Temos nossa primeira *Green Flag* (**Figura 5**).

Ou seja, nosso primeiro teste está passando. Tudo o que esperamos do sistema até agora está funcionando perfeitamente. Vamos então continuar nosso ciclo eliminando as possíveis duplicações que, até agora, são inexistentes. Por isso, fechamos nosso primeiro ciclo partindo para a inclusão de um segundo teste: "Adicionar um número à pilha e verificar se a mesma não está vazia".

Lembrando do mantra (*Red, Green, Refactor*), vamos iniciar implementando o teste. Adicione à seção *published* de seu *test case* o método "TestPushNaoVazia", e implemente-o conforme a **Listagem 5**.

**Listagem 5. Adicionando o método TestPushNaoVazia**

```
procedure TPilhaTests.TestPushNaoVazia;
var
  UmaPilha: TPilha;
begin
  UmaPilha := TPilha.Create;
  UmaPilha.Push(1);
  CheckFalse(UmaPilha.EstaVazia);
  UmaPilha.Free;
end;
```

Se tentarmos compilar, receberemos um erro dizendo que o método *Push* é inexistente. Volte à classe *TPilha* e declare o método *Push* simplesmente:

```
procedure TPilha.Push(Numero: Integer);
begin

end;
```

Nosso teste compilará e agora é hora de vê-lo "não" passando. Isso significa que essa funcionalidade ainda não está implementada. Para fazê-lo passar, apenas coloque o

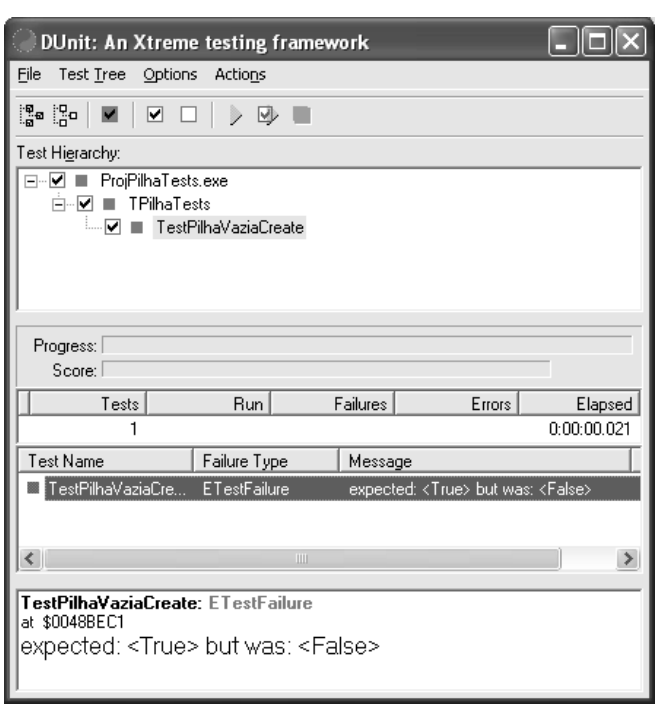



**Figura 4.** Nosso primeiro teste falhando no DUnit

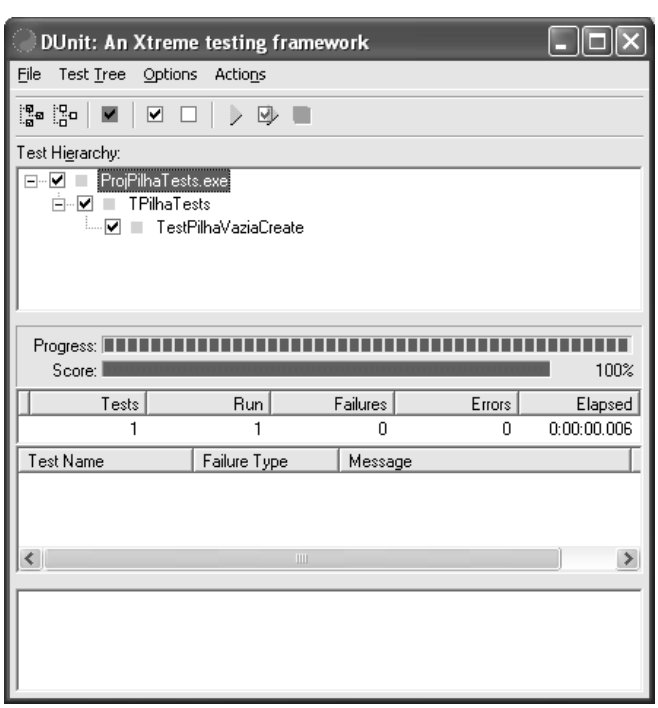



**Figura 5.** Nosso primeiro teste passando no DUnit

seguinte código no corpo do método *Push*:

```
EstaVazia := False;
```

Parece ridículo o que estamos fazendo, mas o teste não está passando? Se queremos dar um comportamento diferente para a nossa classe, temos que escrever outros testes introduzindo novas funcionalidades e não mudar o que já existe sem nenhum motivo.

Chega então a hora de eliminarmos as duplicações e melhorar a estrutura interna de nosso código. Como temos os testes passando, temos a certeza de que as modificações não afetarão o comportamento de nossa classe.

Uma melhoria para a nossa classe seria criar uma propriedade para *EstaVazia*, encapsulando esse campo, em vez de deixá-lo com visibilidade *public*. Altere o código da unit *Pilha* para o da **Listagem 6**.

**Listagem 6. Unit Pilha após a adição do segundo teste**

```
unit Pilha;

interface

type
  TPilha = class
  private
    FEstaVazia: Boolean;
  public
    constructor Create;
    procedure Push(Numero: Integer);
    property EstaVazia: Boolean read FEstaVazia;
  end;

implementation

{ TPilha }

constructor TPilha.Create;
begin
  FEstaVazia := True;
end;

procedure TPilha.Push(Numero: Integer);
begin
  FEstaVazia := False;
end;
end.
```

Lembre-se de rodar os testes para ver que estão passando. Vamos para a eliminação de uma outra duplicação. Nosso test case possui em duplicidade a declaração, a criação e a liberação do ponteiro para a Pilha.

Existem dois métodos conhecidos como SetUp e TearDown, padrões da classe TTestCase, que são executados exatamente antes e após cada teste, respectivamente. Por isso, podemos utilizá-los exatamente nesses casos, para alocação e liberação de recursos, deixando o DUnit cuidar do resto. Troque os trechos de código de seu test case pelos da **Listagem 7**.

**Listagem 7. Test Case após a adição do segundo teste**

```
uses Pilha;

  TPilhaTests = class(TTestCase)
  private
    FPilha: TPilha;
  protected
    procedure SetUp; override;
    procedure TearDown; override;
  public
  published
    procedure TestPilhaVaziaCreate;
    procedure TestPushNaoVazia;
  end;
(...)
procedure TPilhaTests.SetUp;
begin
  inherited;
  FPilha := TPilha.Create;
end;

procedure TPilhaTests.TearDown;
begin
  inherited;
  FPilha.Free;
end;

procedure TPilhaTests.TestPilhaVaziaCreate;
begin
  CheckTrue(FPilha.EstaVazia);
end;

procedure TPilhaTests.TestPushNaoVazia;
begin
  FPilha.Push(1);
  CheckFalse(FPilha.EstaVazia);
end;
(...)
```

Rodando os testes mais uma vez e olhando-os passar, completamos o ciclo do segundo teste.

## Adicionando testes

Vamos então continuar escrevendo mais um teste, dessa vez: "Adicionar um número à pilha e verificar se o mesmo é retornado no Pop". Implemente e declare o teste a seguir:

```
procedure TPilhaTests.TestPushPopNroCerto;
begin
  FPilha.Push(1);
  CheckEquals(1, FPilha.Pop);
end;
```

Não podemos compilar ainda, pois *Pop* não existe. Vamos declará-lo rapidamente na classe *Pilha*:

```
function TPilha.Pop: Integer;
begin

end;
```

Agora o código compila, mas os testes ainda não passam. Para fazer o teste passar, implemente no corpo do método anterior, o código:

```
Result := 1;
```

Com isso você pode compilar e o teste certamente passará. Nenhuma duplicação foi encontrada, é hora de passarmos para o teste: "Adicionar três números e verificar se, com Pop, retornam na ordem inversa".

Talvez esse seja o teste mais importante, pois dirá exatamente como especificar a principal regra que define uma pilha. Adicione e declare o código da **Listagem 8** na unit de testes.

**Listagem 8. Implementando o terceiro teste**

```
procedure TPilhaTests.TestPush3Pop3NrosCertos;
begin
  FPilha.Push(1);
  FPilha.Push(2);
  FPilha.Push(3);

  CheckEquals(3, FPilha.Pop);
  CheckEquals(2, FPilha.Pop);
  CheckEquals(1, FPilha.Pop);
end;
```

Note que aqui estamos utilizando outro método para a asserção. Na verdade, o padrão *xUnit* define uma classe

*Assert* que não é seguida estritamente pelo *DUnit*. Porém, nosso framework possui muitos métodos iniciados com *Check* para os mais diversos tipos de comparações.

Tais procedimentos podem ser facilmente encontrados com suas descrições na documentação da ferramenta. O teste compilará, mas, ao rodá-lo, receberemos *Red Flag*, indicando que o mesmo não passou. Declare então a variável *FValores* na seção *private* da classe *TPilha* da seguinte forma:

```
FValores: TIntegerDynArray;
```

Não esquecendo de adicionar *Types* à seção *uses*, modifique os métodos existentes conforme a **Listagem 9**.

**Listagem 9. Alterando os métodos Push e Pop**

```
procedure TPilha.Push(Numero: Integer);
begin
  SetLength(FValores, Length(FValores) + 1);
  FValores[High(FValores)] := Numero;
  FEstaVazia := False;
end;

function TPilha.Pop: Integer;
begin
  Result := FValores[0];
  SetLength(FValores, Length(FValores) - 1);
end;
```

Compile e rode o teste. Ele ainda não passa. Verifique o código que acabamos de escrever. Estamos criando um *Array* dinâmico para armazenar os valores, inserindo os números no método *Push* e retirando-os em *Pop*.

Porém, no último método não estamos retornando o valor do topo da pilha, mas sim do início. Esse é um erro que poderia ter acontecido com qualquer um e, pela primeira vez, nossos testes nos avisaram de que algo está errado! Corrija então a sentença para:

```
Result := FValores[High(FValores)];
```

Agora nosso teste passará como esperado e nossa pilha já está tomando forma. Nesse momento, fica como tarefa para o leitor implementar os próximos testes e, por conseqüência, os métodos da classe *TPilha*. Apenas ressalto mais um que considero muito importante: "Tentar retirar com Pop quando a Pilha está vazia".

Quando uma situação dessas acontece, o que devemos esperar? Uma exceção? Isso na verdade, quem vai definir é o programador através do seu teste. Se a especificação diz que a classe não deve disparar erros desse tipo, então esperaremos que nenhuma exceção ocorra, caso contrário, devemos verificar se a mesma foi lançada.

Não verificar situações de erro (os chamados *Negative Cases*) é muito comum entre os programadores e testadores, mas em nosso caso (TDD), isso não pode acontecer, pois só escreveremos código para disparar erros se houver um teste indicando isso.

Uma maneira clássica para testar exceções é a que utilizaremos. Porém, existem frameworks como o *NUnit* (para .NET) muito melhor preparados para tal situação. Digite então o teste, conforme a **Listagem 10**.

**Listagem 10. Teste para verificar valores**

```
procedure TPilhaTests.TestPopSemValores;
var
  ErroAconteceu: Boolean;
begin
  ErroAconteceu := False;
  try
    FPilha.Pop;
except
    ErroAconteceu := True;
  end;
  CheckTrue(ErroAconteceu);
end;
```

Rodando o teste, veremos que o mesmo passará de primeira, o que não está errado. Isso apenas confirma que tal funcionalidade já foi implementada. Poderíamos também esperar outro tipo de exceção ou determinada mensagem. Para isso, deveríamos modificar primeiramente nosso teste. Isso depende logicamente de cada caso.

Ao implementar os outros testes, você deverá achar algumas duplicações entre os métodos *Top* e *Pop* que podem ser resolvidas com refatoração. Fica aí um bom exercício para treinar todas as técnicas apresentadas até o momento.

## TDD na prática

Neste artigo, vimos um exemplo meramente didático do que pode ser feito utilizando-se essa metodologia de desenvolvimento. No entanto, qualquer tipo de classe pode

ser desenvolvida dessa maneira, umas com mais facilidade, outras não.

Existem partes mais difíceis de testar com *unit tests* e, por isso, não são fáceis de serem elaboradas com TDD. Algumas regiões mais críticas são:

- Classes com acesso a recursos externos, como HD, socket ou bancos de dados;
- Interface Gráfica.

Para realizar testes em classes que acessão recursos externos, utilizamos uma técnica avançada conhecida como *Mock Objects*, algo de muita discussão em fóruns e páginas da internet. Já para interfaces gráficas, geralmente a cobertura é feita através de testes de aceitação, pois também existem várias ferramentas para tanto.

Para testes em ASP.NET, você pode testar a camada de interface utilizando uma ferramenta conhecida como *NUnitASP*, também bastante divulgada na internet. Caso tenha alguma dificuldade, tente iniciar sua busca em páginas específicas de *Test-Driven Development* ou ferramentas de busca.

## Conclusão

Hoje não podemos mais pensar em desenvolver sistemas sem investir também em qualidade. O TDD, além de garantir uma quantidade muito baixa de *bugs*, com seu *mantra*, ajuda a melhorar o *design* de nosso modelo, trazendo inúmeros benefícios.

Em um projeto de software, geralmente estimamos um prazo e custo para a elaboração do programa propriamente dito e outro para a correção de erros. No *Test-Driven Development*, devido ao fato de possuirmos um volume de erros bastante controlado, podemos estimar muito mais precisamente prazos e custos.

Tente exercitar um pouco e comece a escrever algumas classes utilizando a metodologia. Faça seu diferencial e saia na frente. Em pouco tempo, você pegará experiência e não conseguirá mais elaborar uma classe sem pensar no nível de qualidade do seu código.

E lembre-se sempre: *Red*, *Green*, *Refactor*! ■

**Links**

**Principal site sobre Test-Driven Development existente na internet**
*www.testdriven.com*

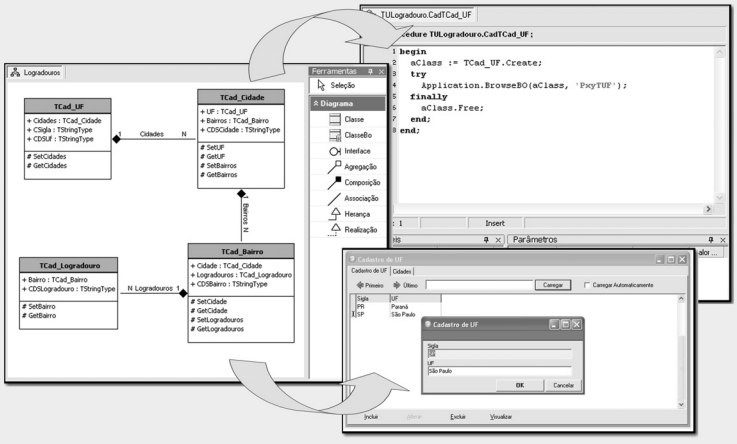
**Site da ferramenta DUnit**
*dunit.sourceforge.net*

**Site de uma grande celebridade na área da programação atual**
*www.martinfowler.com*

**Um dos sites mais conhecidos sobre eXtreme Programming**
*www.xprogramming.com*


Faça do tempo seu diferencial

Ambiente de Análise e Implementação integrados.
Interface Gráfica e Banco de Dados gerados a partir do modelo.



Persistência com:
InterBase
Firebird
MySQL
MS SQL Server

Compatível com:
Turbo Delphi
Delphi
Kylix

Desenvolva para:
Windows
Linux

Conheça a versão FREE!

Speed CASE™

Ferramenta de produtividade para desenvolvimento de software

www.speedcase.com.br

Fone: (44) 3225-7788

JORGE LUIS BUBLITZ

# Uma abordagem prática de Stored Procedures e Triggers

**JORGE LUIS BUBLITZ**
(bublitz@hotmail.com), é desenvolvedor autodidata desde os 14 anos. Começou com BASIC nos micros de 8 bits, como o Apple II (TK-2000), TRS-80 (CP-500) e MSX (Expert). Desenvolveu em Clipper e em Turbo Pascal desde a versão 3. Desenvolve em Delphi desde a versão 1. Atualmente é funcionário concursado do Tribunal Regional Eleitoral de Mato Grosso, onde coordena e desenvolve diversos projetos Cliente/Servidor, Multicamadas, .NET e VoIP, todos em Delphi.

Quando começamos o desenvolvimento de um sistema, algumas perguntas que não querem se calar: "Quais regras de negócio ficarão no banco de dados? Quais devem ficar na camada servidora? E quais ficariam na camada cliente?". Responder a essas três perguntas não é fácil. Se você coloca todas as regras no banco, corre o risco de sobrecarregá-lo. O mesmo vale para a aplicação servidora, seja ela cliente/servidor ou multicamadas. E no cliente? Devo colocar alguma regra?

Existem algumas sugestões básicas: quando você desenvolve uma aplicação Web, as regras mais simples você utiliza *JavaScript* para checar/validar no próprio navegador, não é mesmo? Campos obrigatórios, validação de datas, verificação de CPF e outras.

E as outras? Como decidir quais ficarão na aplicação servidora e quais ficarão no banco? Eu uso a seguinte lógica: se a regra envolve atualização em uma ou duas tabelas e com isso incluir/alterar um ou dois registros, prefiro deixá-la na camada da aplicação, pois muito provavelmente serão enviados um ou dois comandos SQL para o banco pela rede.

Agora, se a regra atualiza mais de duas tabelas e com isso temos que incluir/alterar vários registros, então é forte candidata a ficar no banco. Há também aquelas regras que mudam com muita freqüência. Prefira deixá-las no banco.

Obviamente que isso é muito relativo. Por exemplo: se o cliente tem um servidor de banco de dados dedicado, você pode colocar mais regras no banco, pois a máquina vai agüentar a carga de serviço. O melhor caminho mesmo é testar.

E é justamente isso que faremos. Partindo de um sistema semi pronto, com uma solução, digamos, pouco aconselhável de uma regra de negócio, vamos primeiramente colocar essa regra de maneira mais segura na própria aplicação. Depois vamos colocar a mesma regra no banco e assim poderemos analisar qual a melhor solução. Utiliza-

remos Delphi com o banco de dados Firebird. Sinta-se a vontade em utilizar outro banco de dados. Aliás, estou testando a versão *Express* do *DB2* da *IBM* e estou gostando muito.

**Nota:** A criação de banco de dados no Firebird, bem como detalhes sobre configurações de conexões dbExpress, estão fora do escopo deste artigo. Você pode encontrar muitos artigos no portal, acessando *www.clubedelphi.net/portal*.

## O sistema de exemplo

A aplicação CDEstoque (acho que já vi esse nome em algum lugar) é bastante simples. O modelo de dados está na **Figura 1**.

Não é a intenção aqui desenvolver uma aplicação completa, passo a passo. Por esse motivo algumas tabelas estão resumidas, com poucos campos, especialmente as tabelas de fornecedores e de clientes, e outras nem estão no modelo, como vendedores.

A aplicação tem o seguinte funcionamento: o cliente primeiramente faz um orçamento dos produtos que pretende comprar, inclusive com a forma de pagamento. O orçamento fica gravado no banco para posterior confirmação (**Figura 2**). Quando o orçamento é confirmado, o sistema grava os dados na tabela de venda, baixa o estoque, gera as parcelas a receber e grava a movimentação dos produtos.

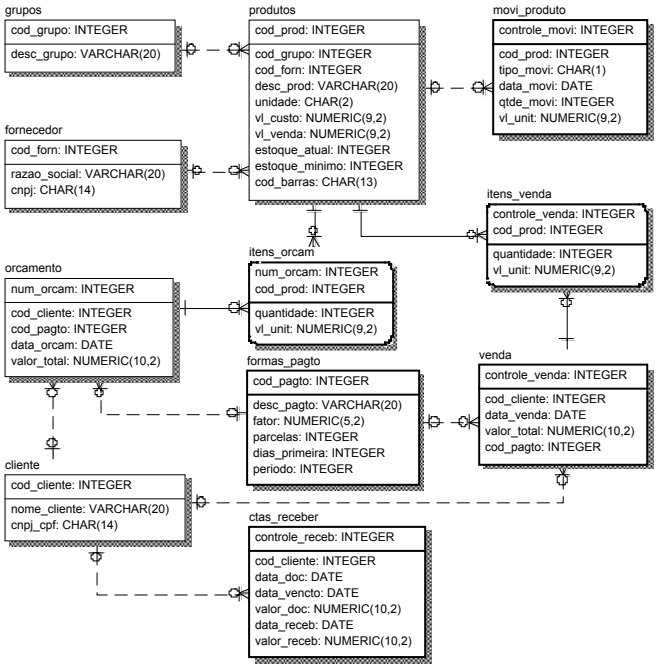



**Figura 1.** Modelo de Dados do CDEstoque

## Primeira versão

Nessa primeira versão, o desenvolvedor colocou a regra de confirmação do orçamento na unit do formulário de orçamento (*untOrcam*), que é um lugar inapropriado (**Listagem 1**).

Também não faz nenhum controle de erro (*try...except*), e não utiliza transação, o que é perigoso, pois se no meio do processo acontecer algum problema, queda de energia, travamento da máquina, a rede cair e outros, as tabelas poderão ficar incompletas, com dados inconsistentes. Podemos ter a situação de baixar os produtos e não gerar as contas a receber, por exemplo. O código está bem comentado, não necessitando de maiores detalhes.

Analisando a rotina de confirmação de orçamentos, vemos que temos duas estruturas de *loop*: uma para lançar os itens do orçamento na tabela de venda, atualizar o estoque e lançar o movimento e outro para gerar as parcelas a receber.

É justamente nesses dois pontos que temos o maior problema. Imagine um orçamento com 50 itens e com uma forma de pagamento em 12 parcelas. Vamos calcular quantos comandos SQL serão gerados:

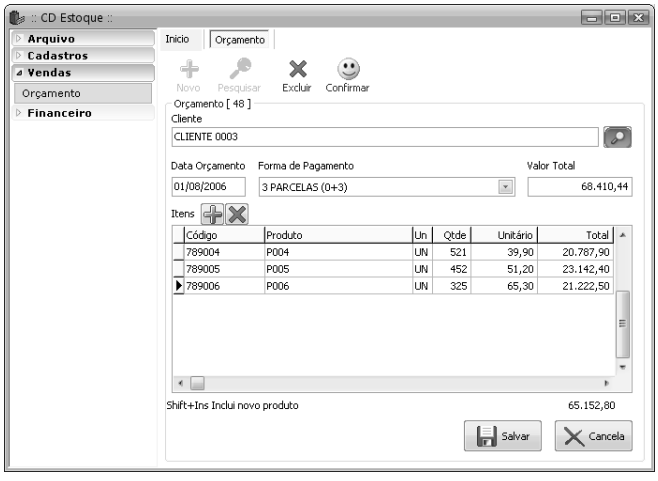



**Figura 2.** A aplicação sendo executada


+ de 80.000 membros cadastrados
+ de 15.000 exemplos com fontes
+ de 900 apostilas
+ de 4.000 dicas
Fórum Delphi
Artigos
TOTALMENTE GRÁTIS
www.delphi.eti.br
Um dos maiores sites de apoio a desenvolvedores Delphi do Brasil!!!

**Listagem 1. Primeira versão da regra de negócio**

```
procedure TfrmOrcam.actConfirmaExecute(
  Sender: TObject);
var
  i, ControleVenda: integer;
  val_parcela: Currency;
  data_vencto: TDate;
begin
  ActiveControl := nil;
  if Application.MessageBox('Confirma o orçamento??',
    'Confirmação', MB_ICONINFORMATION +
    MB_YESNO) = IDYES then
  begin
    with dm do
    begin
      if cdsItemOrcam.State in [dsEdit, dsInsert] then
        cdsItemOrcam.Post;
      cdsOrcam.Post;
      if cdsOrcam.ChangeCount > 0 then
        cdsOrcam.ApplyUpdates(0);
      { Função no DataModule }
      ControleVenda := NextVenda;

      cdsVenda.Open;
      cdsItemVenda.Open;
      cdsMovi.Open;
      cdsProduto.Open;
      cdsReceb.Open;

      { Grava Dados do Orçamento na tabela Vendas }
      cdsVenda.Append;
      cdsVendaCONTROLE_VENDA.Value := ControleVenda;
      ...
      cdsVenda.Post;
      cdsVenda.ApplyUpdates(0);

      cdsItemOrcam.First;
      while not cdsItemOrcam.Eof do
      begin
        { Grava Itens do Orçamento na tabela
         Itens de Venda }
        cdsItemVenda.Append;
        cdsItemVendaCONTROLE_VENDA.Value :=
          ControleVenda;
...
        cdsItemVenda.Post;

        { Lança dados de Movimentação }
        cdsMovi.Append;
        cdsMoviCOD_PROD.Value :=
          cdsItemOrcamCOD_PROD.Value;
        ...
        cdsMovi.Post;

        { Atualiza Estoque }
        cdsProduto.Close;
        sqlProduto.Params[0].asInteger  :=
          cdsItemOrcamCOD_PROD.Value;
        cdsProduto.Open;
        cdsProduto.Edit;
        ...
        cdsProduto.Post;

        cdsItemOrcam.Next;
      end;
      cdsItemVenda.ApplyUpdates(0);
      cdsMovi.ApplyUpdates(0);
      cdsProduto.ApplyUpdates(0);

      { Lança Contas a Receber }
      val_parcela :=
        cdsOrcamValorFinal.AsCurrency/cdsFPagtoPARCELAS.Value;
      data_vencto := IncDay(Date,
        cdsFPagtoDIAS_PRIMEIRA.Value);
      for i := 1 to cdsFPagtoPARCELAS.Value do
      begin
        cdsReceb.Append;
        ...
        cdsReceb.Post;
        data_vencto := IncDay(data_vencto,
          cdsFPagtoPERIODO.Value);
      end;
      cdsReceb.ApplyUpdates(0);

      { Exclui Orçamento }
      cdsOrcam.Delete;
      cdsOrcam.ApplyUpdates(0);

      ...
    end;
    ShowMessage('Orçamento confirmado!!!');
  end;
end;
```

**1:** incluir os dados do orçamento na tabela de vendas;

**50:** incluir os itens do orçamento na tabela de itens de venda;

**50:** incluir a movimentação do produto (*Movi_produto*);

**50:** atualizar o estoque;

**12:** incluir as parcelas no contas a receber.

Total de **163** comandos SQL indo e vindo pela rede.

## Segunda versão

Agora vamos ver uma solução mais segura, no código do DataModule (*untDatMod*) da **Listagem 2.**

**Listagem 2. Regra de negócio no DataModule**

```
type
  TModoTrans = (mtInicia, mtGrava, mtCancela);
  Tdm = class(TDataModule)
  ...
    procedure Transac(ModoTrans: TModoTrans);
    function ConfOrcam: boolean;
  private
    TransDesc: TTransactionDesc;
  public
    CodFPagto: integer;
  end;
  ...

procedure Tdm.DataModuleCreate(Sender: TObject);
...
  TransDesc.TransactionID := 1;
  TransDesc.IsolationLevel := xilREADCOMMITTED;
...

procedure Tdm.Transac(ModoTrans: TModoTrans);
begin
  case ModoTrans of
    mtInicia:
      { CDEstoque é o nome do SQLConnection }
      if not CDEstoque.InTransaction then
        CDEstoque.StartTransaction(TransDesc);
    mtGrava:
      try
        if CDEstoque.InTransaction then
          CDEstoque.Commit(TransDesc);
      except
        CDEstoque.Rollback(TransDesc);
      end;
    mtCancela:
      if CDEstoque.InTransaction then
        CDEstoque.Rollback(TransDesc);
  end;
end;

function Tdm.ConfOrcam: Boolean;
var
  i, ControleVenda: integer;
  val_parcela: Currency;
  data_vencto: TDate;
begin
  { Inicia transação }
  Transac(mtInicia);
  try
    { Função no DataModule }
    ControleVenda := NextVenda;

    ... { Código igual ao que tinha no untOrcam }
```

```
    { Exclui o Orçamento }
    cdsOrcam.Delete;
    cdsOrcam.ApplyUpdates(0);

    { Grava a transação }
    Transac(mtGrava);

    cdsVenda.Close;
    cdsItemVenda.Close;
    cdsMovi.Close;
    cdsProduto.Close;
    cdsReceb.Close;

    Result := True;
  except
    { Se algo saiu errado cancela a transação }
    Transac(mtCancela);
    Result := False;
  end;
end;
```

O código na *untOrcam* foi alterado de acordo com a **Listagem 3**.

**Listagem 3. Modificando o formulário para a segunda versão da regra de negócio**

```
procedure TfrmOrcam.actConfirmaExecute(
  Sender: TObject);
begin
  ActiveControl := nil;
  if Application.MessageBox('Confirma o orçamento??',
    'Confirmação', MB_ICONINFORMATION +
    MB_YESNO) = IDYES then
  begin
    with dm do
    begin
      if cdsItemOrcam.State in [dsEdit, dsInsert] then
        cdsItemOrcam.Post;
      cdsOrcam.Post;
      if cdsOrcam.ChangeCount > 0 then
        cdsOrcam.ApplyUpdates(0);

      if ConfOrcam then
        ShowMessage('Orçamento confirmado!!!')
      else
        ShowMessage(
          'Erro!! Orçamento não confirmado!!!');
    end;
  end;
end;
```

Nessa segunda versão, temos um exemplo mais bem organizado e mais seguro da nossa regra de negócio. Procure sempre deixar no DataModule as regras de banco. Primeiro declaramos um tipo (*TModoTrans*), que será utilizado na procedure para iniciar, gravar ou finalizar transações.

Depois declaramos a procedure de transação (*Transac*) e a função para confirmar orçamento (*ConfOrcam*). Em seguida declaramos uma variável (*TransDesc*) do tipo *TTransactionDesc*, que é utilizada para definir e controlar a transação em nossa aplicação. No dbExpress, *TtransactionDesc* é uma estrutura (record) que define a transação. Um ponteiro dessa estrutura é passado cada vez que se inicia, finaliza ou cancela uma transação.

Na criação do DataModule definimos o ID da transação e o nível de isolamento. O ID que declaramos é para essa aplicação. Se você precisar de uma outra transação na mesma aplicação, declare outra variável e defina outro ID para ela.

A função de confirmação de pedidos é a mesma da primeira versão, só que agora está dentro da estrutura de erro (*try...except*) e também dentro de uma transação, assim temos a certeza que todas as alterações no banco serão gravadas.

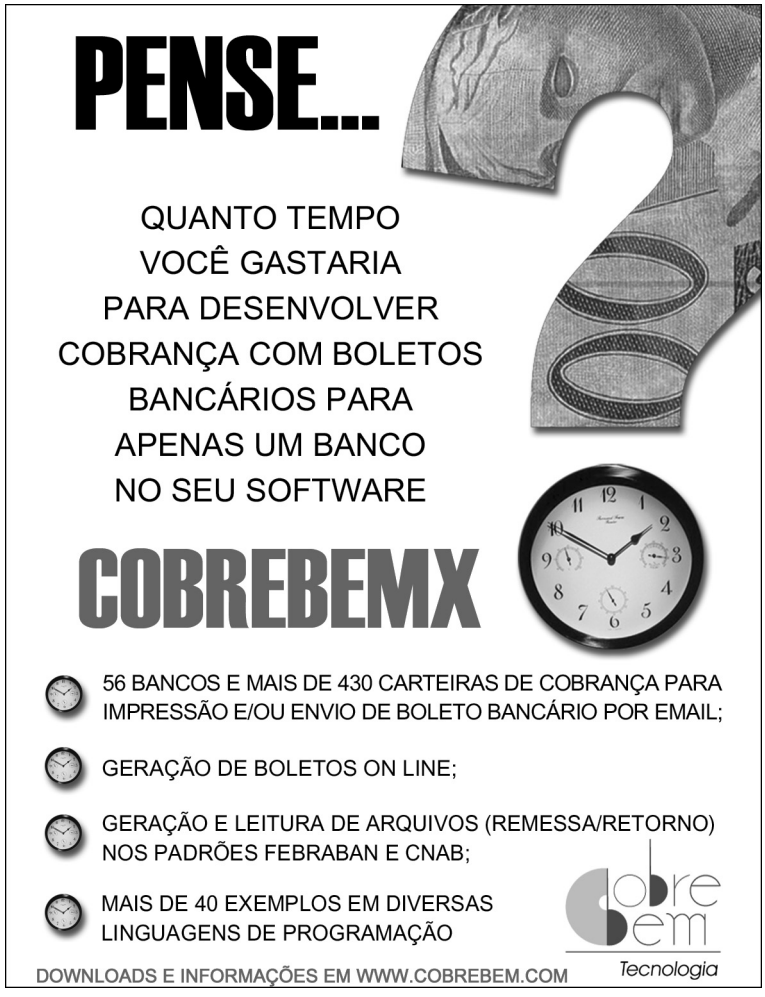
Mas a quantidade de comando SQL sendo enviados pela rede não mudou. Então, seguindo a lógica do início do artigo, vamos colocar essa regra de negócio no banco.

## Terceira versão

A maior vantagem do uso de *Stored Procedures* e *Triggers* é a redução no tráfego de rede, já que são executadas pelo banco na máquina servidora. Outro aumento de desempenho é no fato que cada vez que a aplicação cliente envia um comando SQL para o servidor, o comando tem que ser processsado, ou seja, analisado gramaticalmente, submetido ao otimizador para formulação de um plano de execução.

*SP's* e *Triggers* são analisadas, otimizadas e armazenadas em uma forma executável no momento em que são adicionadas ao banco de dados. A partir do momento que uma *SP/Trigger* não tem que ser analisada e otimizada cada vez que é chamada, ela é executada mais rapidamente que um comando SQL equivalente.

A manutenção também é facilitada porque você pode alterar uma *SP/Trigger* sem ter que alterar ou mesmo recompilar as aplicações que acessam o banco (desde que os parâmetros não sofram alterações). Há também um aumento

**Listagem 4. Triggers e Stored Procedure da regra de negócio da aplicação**

```
/* Trigger para Incluir dados no movimento*/
CREATE TRIGGER INCLUI_MOVIMENTO FOR ITENS_VENDA
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  INSERT INTO MOVI_PRODUTO
  (COD_PROD, TIPO_MOVI, DATA_MOVI, QTDE_MOVI, VL_UNIT)
  VALUES
  (NEW.COD_PROD, 'S', CURRENT_DATE, NEW.QUANTIDADE,
   NEW.VL_UNIT);
END

/* Trigger para atualizar estoque */
CREATE TRIGGER ATUALIZA_ESTOQUE FOR MOVI_PRODUTO
ACTIVE BEFORE INSERT POSITION 0
AS
DECLARE VARIABLE QTDE INTEGER;
BEGIN
  IF (NEW.TIPO_MOVI = 'S') THEN
    QTDE = -NEW.QTDE_MOVI;
  ELSE
    QTDE = NEW.QTDE_MOVI;

  UPDATE PRODUTOS SET
  ESTOQUE_ATUAL = ESTOQUE_ATUAL + :QTDE
  WHERE COD_PROD = NEW.COD_PROD;
END

/* Procedure para confirmar o Orcamento */
CREATE PROCEDURE CONF_ORCAM (
    NUM_ORCAM INTEGER)
AS
DECLARE VARIABLE CONT_VENDA INTEGER;
DECLARE VARIABLE COD_PAGTO INTEGER;
DECLARE VARIABLE VALOR_TOTAL NUMERIC(10,2);
DECLARE VARIABLE NUM_PARCELAS INTEGER;
DECLARE VARIABLE DIAS_PRIMEIRA INTEGER;
DECLARE VARIABLE PERIODO INTEGER;
DECLARE VARIABLE DATA_VENCTO DATE;
DECLARE VARIABLE VALOR_PARCELA INTEGER;
DECLARE VARIABLE I INTEGER;
DECLARE VARIABLE COD_CLIENTE INTEGER;
BEGIN
  /* Pega o número da próxima venda */
  CONT_VENDA = GEN_ID(GEN_VENDA_ID, 1);

  /* Inclui dados do orçamento em vendas */
  INSERT INTO VENDA
  (CONTROLE_VENDA, COD_CLIENTE, DATA_VENDA,
   VALOR_TOTAL, COD_PAGTO)
  SELECT :CONT_VENDA, COD_CLIENTE, CURRENT_DATE,
   VALOR_TOTAL, COD_PAGTO
   FROM ORCAMENTO
   WHERE NUM_ORCAM = :NUM_ORCAM;

  /* Inclui itens do orçamento em itens_vendas */
  INSERT INTO ITENS_VENDA
  (CONTROLE_VENDA, COD_PROD, QUANTIDADE, VL_UNIT)
  SELECT :CONT_VENDA, COD_PROD, QUANTIDADE, VL_UNIT
   FROM ITENS_ORCAM
   WHERE NUM_ORCAM = :NUM_ORCAM;

  /* Pega o código de pagamento, valor total e o
    código do cliente */
  SELECT COD_PAGTO, VALOR_TOTAL, COD_CLIENTE
  FROM ORCAMENTO
  WHERE NUM_ORCAM = :NUM_ORCAM
  INTO :COD_PAGTO, :VALOR_TOTAL, :COD_CLIENTE;

  /* Pega o número de parcelas, dias para a primeira
    parcela e periodo entre as parcelas */
  SELECT PARCELAS, DIAS_PRIMEIRA, PERIODO
  FROM FORMAS_PAGTO
  WHERE COD_PAGTO = :COD_PAGTO
  INTO :NUM_PARCELAS, :DIAS_PRIMEIRA, :PERIODO;

  /* Inclui parcelas */
  I = 0;
  DATA_VENCTO = CURRENT_DATE + :DIAS_PRIMEIRA;
  VALOR_PARCELA = :VALOR_TOTAL / :NUM_PARCELAS;
  WHILE (I < :NUM_PARCELAS) DO
  BEGIN
    INSERT INTO CTAS_RECEBER
    (COD_CLIENTE, DATA_DOC, DATA_VENCTO, VALOR_DOC)
    VALUES
    (:COD_CLIENTE, CURRENT_DATE, :DATA_VENCTO, :VALOR_PARCELA);
    I = I + 1;
    DATA_VENCTO = :DATA_VENCTO + :PERIODO;
  END

  /* Exclui orçamento */
  DELETE FROM ORCAMENTO
  WHERE NUM_ORCAM = :NUM_ORCAM;
END
```

de segurança uma vez que as *SP/Trigger* podem acessar tabelas que o usuário não tem direito de fazê-lo.

Por exemplo, suponha que um usuário precise rodar um relatório que mostra o total de vendas de um período. Embora essas informações venham da tabela de vendas, você não quer que o usuário tenha acesso às vendas individuais de cada vendedor.

A solução é fazer uma *SP* que emita essa informação resumida. Você pode então dar direito ao usuário de executar a *SP* e ele não precisa ter direitos sobre a tabela de vendas. Vamos criar as *SP/Triggers* para essa regra de negócio.

Teremos uma *Stored Procedure* que receberá como parâmetro no número do orçamento. Ela fará a inclusão dos dados do orçamento na tabela *Venda*, incluirá os itens e gerará as parcelas a receber. A tabela *Itens_Venda* terá uma *Trigger* no evento *Before Insert* que incluirá a movimentação na tabela *Movi_Produto*. Essa por sua vez terá também outra *Trigger* que atualizará o estoque. Na **Listagem 4** temos o código das duas *Triggers* bem como o código da *Stored Procedure*.

Vamos alterar agora a aplicação. No DataModule adicionei um *SQLStoredProc* (*dbExpress*), alterando a propriedade *SQLConnection* para *CDEstoque* (nome do *SQLConnection* do DataModule) e *StoredProcName* para CONF_ORCAM. O código da *ConfOrcam* ficará semelhante ao da **Listagem 5**. Vejam como ficou bem simples o código.

**Listagem 5. Procedure ConfOrcam no DataModule**

```
function Tdm.ConfOrcam: Boolean;
begin
  { Inicia a transação }
  Transac(mtInicia);
  try
    spConfOrcam.Params[0].AsInteger :=
      cdsOrcamNUM_ORCAM.Value;
    spConfOrcam.ExecProc;
    { Grava a transação }
    Transac(mtGrava);
  except
    { Se algo saiu errado, cancela a transação }
    Transac(mtCancela);
    Result := False;
  end;
end;
```

## Conclusão

O uso correto de *Stored Procedures/Triggers* traz uma série de vantagens, onde podemos destacar a redução no tráfego de rede e o aumento de performance e escalabilidade das suas aplicações. Abraços e boas compilações! ■

### CLUBEDELPHI PLUS!

Acesse agora o mesmo o portal do assinante ClubeDelphi (www.clubedelphi.net/portal) e assista a uma vídeo-aula de Luciano Pimenta, que mostra como trabalhar com Triggers e Stored Procedures para regras de negócio no banco, do mini curso de aplicações Client/Server com dbExpress e Firebird

*http://www.devmedia.com.br/articles/viewcomp.asp?comp=2083*

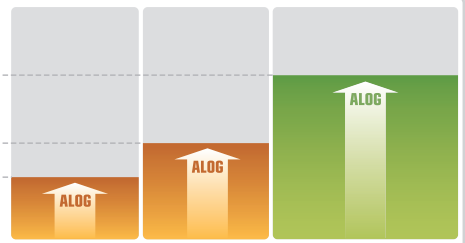
# FAÇA COMO A ALOG
# FOQUE EM SEU NEGÓCIO

A ALOG é totalmente focada em serviços de Data Center. Dedica-se 24 horas por dia, 7 dias por semana a cuidar do negócio dos nossos 320 clientes corporativos.

ALOG desenvolveu no Brasil o conceito de **HOSTING GERENCIADO**: um serviço de excelência que garantirá a sua empresa estabilidade, agilidade, redundância, e escalabilidade. Para que isso seja verdade os clientes da ALOG contam com **Suporte Incondicional***, um conceito revolucionário em atendimento no Brasil.

## ENTENDA O **HOSTING GERENCIADO**

| CAMADAS DE NEGÓCIO | DESCRIÇÃO |
|---|---|
| Processo de Negócio | Atividades cotidianas do cliente |
| Aplicação do Cliente | Softwares de Gestão (Ex: ERP, CRM, outros) |
| Ambiente da Aplicação | .NET, JAVA, SQL Server, MySQL, Oracle, Exchange |
| Sistema Operacional | Windows, Linux, Unix, Solaris |
| Servidor ou equipamento | Servidores RISC/CISC, Appliances, Roteadores, Switches |
| Rede | IP, Wan, Rede Local |
| Infra-estrutura Básica | Energia, HVAC, Segurança - 2N+1 |




| ESTRATÉGIA DO CLIENTE | COLOCATION | HOSTING | HOSTING GERENCIADO |
|---|---|---|---|

## PLANOS DE **HOSTING GERENCIADO**

### GERENCIADO 1

- **Pentium D 2.8 GHz**
- 512 RAM DDR2
- 2 HD SATA 80 GB
- 450 GB de Transf. Mensal
- Firewall e Backup

**BOOT REMOTO**
**GERENCIAMENTO****

**R$1.050,00** mensais

**INSTALAÇÃO IMEDIATA*****

### GERENCIADO 2

- **Dual Xeon Dual Core 3.0 GHz**
- 1GB RAM DDR2 533
- 2 HD SATA RAID I 300 GB
- 1.000 GB de Transf. Mensal
- Firewall e Backup

**BOOT REMOTO**
**GERENCIAMENTO****

**R$2.850,00** mensais

**INSTALAÇÃO IMEDIATA*****

ALOG data centers

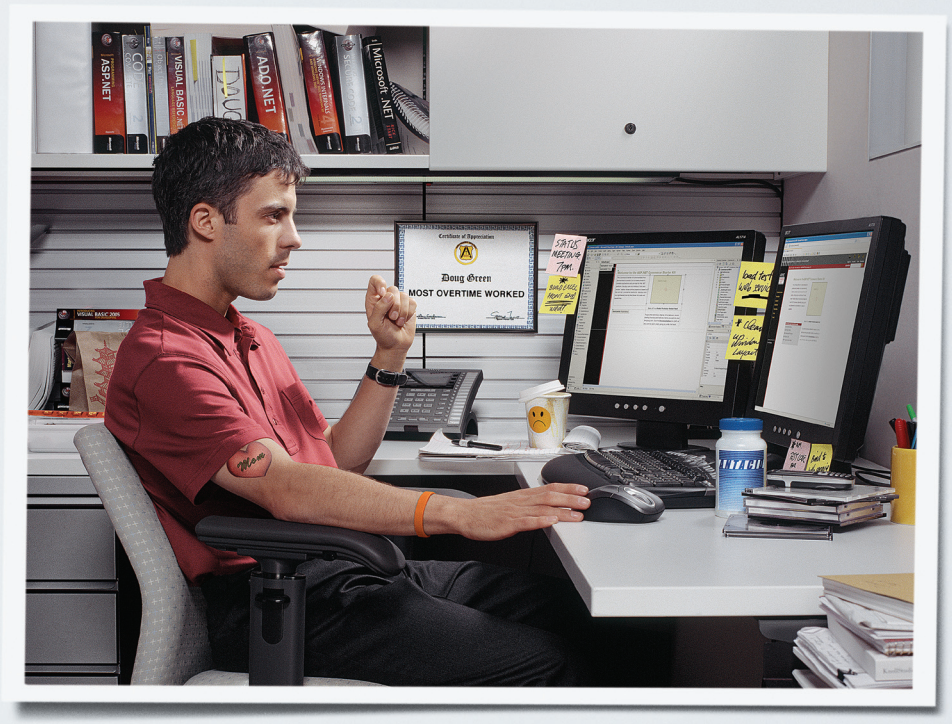
**Excelência em Hosting**
**www.alog.com.br :: 0800 282 3330**

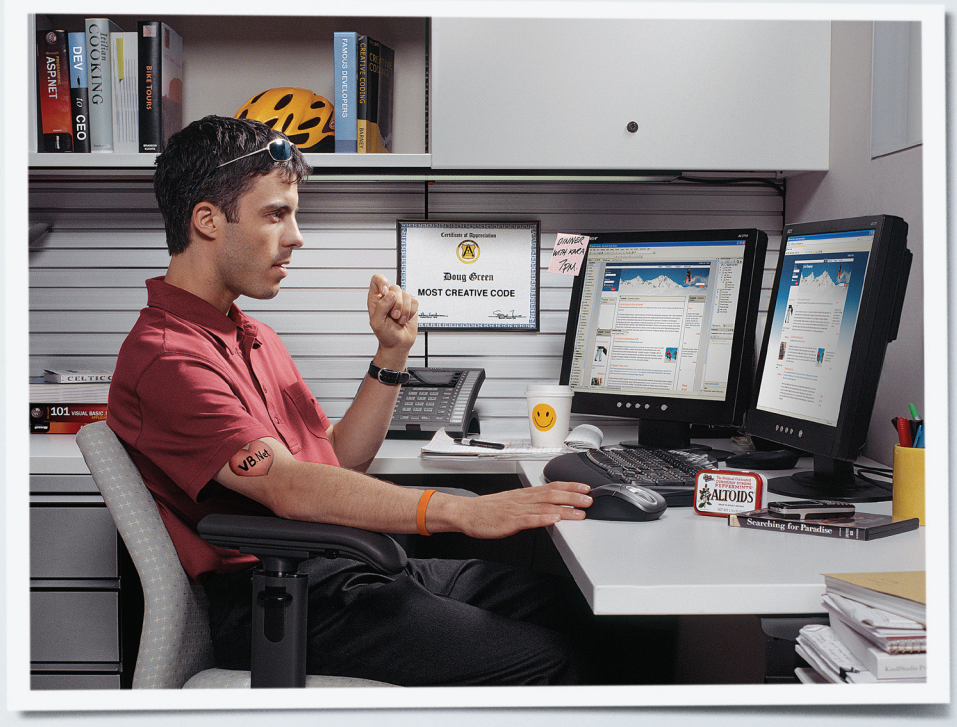
Hosting Gerenciado | Colocation | Contingência | Email Corporativo | Conectividade | Serviços Profissionais

*Seu potencial. Nossa inspiração.*™

Microsoft®

McCANN




# **Novo Visual Studio® 2005.** A diferença é óbvia.

**Descobriu a diferença entre as duas imagens acima?** Quando começar a desenvolver, você verá. O novo Visual Studio® 2005 possui mais de 400 novos recursos que o ajudarão a construir aplicações Web e acelerar seu projeto em plataforma .Net, reduzindo o tempo perdido em tarefas entediantes e repetitivas. Assim, você poderá trabalhar na criação de códigos de qualidade. Saiba mais sobre o novo Visual Studio® 2005 acessando **www.msdn.com.br/visualstudio2005**

Microsoft® Visual Studio® 2005

© 2006 Microsoft Corporation. Todos os direitos reservados. Microsoft, Visual Studio, o logo do Visual Studio e "Seu potencial. Nossa inspiração" são marcas registradas ou marcas da Microsoft Corporation nos Estados Unidos e/ou em outros países.